MaxonC++

Das Entwicklungssystem zur objektorientierten Programmierung

AMIGA

AMIGA

MaxonC++

Das C++ Entwicklungssystem

MAXON
computer

MaxonC++

Autoren:

C/C++ Compiler	**Jens Gelhar**
Edward	Thomas E. Wieger
Maxon Debugger	Christian Spreuer
MakeAPP/RCT.Library	Jürgen Haage
MaxonASM	Klaus-Dieter Sommer
HotHelp	Hartmut Stein

© 1992 by MAXON Computer

Alle Rechte vorbehalten. Dieses Handbuch und die dazugehörige Software ist urheberrechtlich geschützt. Es darf in keiner Form (auch auszugsweise) mittels irgendwelcher Verfahren reproduziert, gesendet, vervielfältigt bzw. verbreitet oder in eine andere Sprache übersetzt werden.

Bei der Erstellung der Programme, der Anleitungen sowie Abbildungen wurde mit allergrößter Sorgfalt vorgegangen. Trotzdem können Fehler nicht ausgeschlossen werden. MAXON übernimmt keinerlei Haftung für Schäden, die auf eine Fehlfunktion von Programmen zurückzuführen sind.

Alle Informationen, die in der vorliegenden Anleitungen enthalten sind, werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Ebenso werden Warenzeichen ohne Gewährleistung einer freien Verwendung benutzt.

MaxonC++ ist da!

Die Welt der professionellen Softwareentwicklung erlebt eine stille Revolution: Das Konzept der objektorientierten Programmierung setzt sich in allen Bereichen durch.

Nachdem Simula in den späten 60er Jahren ihrer Zeit zu weit voraus war und Smalltalk später zu sehr mit allen vertrauten Konzepten der Programmierung brach, um sich durchsetzen zu können, gibt es jetzt endlich eine objektorientierte Programmiersprache, die sich als der neue Standard in der Softwareentwicklung etabliert - die Rede ist natürlich von C++.

Es gibt viele Gründe, die gerade für C++ sprechen. Zunächst setzt diese Sprache auf C auf, der im professionellen Bereich bisher verbreitetsten Sprache. Das geht so weit, daß C-Quelltexte sich mit minimalen oder ganz ohne Modifikationen mit einem C++-Compiler übersetzen lassen. Diese Abwärtskompatiblität erleichtert natürlich den Umstieg von C nach C++ gewaltig und macht C++ für jeden C-Programmierer zur ersten Wahl.

Ferner wurde C++ konsequent so konzipiert, daß eine objektorientierte Lösung weder beim Speicherplatz noch in Bezug auf die Laufzeit einer konventionellen Lösung unterlegen ist und so die Effizienz, die man von C gewohnt ist, erhalten bleibt.

Nicht zuletzt ist zu erwähnen, daß C++ von den AT&T Bell Laboratories entwickelt wurde und von AT&T, also gewiß keiner kleinen Firma, gefördert wird. Schon allein dieser Umstand hat für die Verbreitung von C++ im UNIX-Bereich und bei den sich mehr und mehr am Workstation-Markt orientierenden Microcomputer gesorgt.

Nur in der Amiga-Welt war von all diesen aufregenden Entwicklungen lange Zeit nichts zu spüren. Während auf anderen Systemen sogar jeder bessere C-Compiler objektorientierte Konzepte von C++ übernahm oder gleich zu einem vollständigen C++-System ausgebaut wurde, empfand der Amiga-User schon das Erscheinen von ANSI-kompatiblen C-Compilern als Fortschritt. Da leistungsfähige Compiler aber die Voraussetzung für professionelle Programmentwicklung sind, litt unter diesem Mangel nicht nur der persönliche Komfort der Amiga-Programmierer, sondern auch die Entwicklung der Amiga-Software allgemein wurde behindert und drohte den Anschluß an konkurrierende Systeme endgültig zu verlieren. Es war höchste Zeit, daß etwas geschah.

Nach mehr als fünf Mannjahren Entwicklungszeit und einer umfangreichen Debugging-Phase ist Maxon C++ jetzt einsatzbereit. Maxon C++ ist keine Portierung von einem anderen System, sondern wurde speziell für den Amiga (und auf einem Amiga) entwickelt und auf die Besonderheiten dieses Systems zugeschnitten. Der Compiler ist zum „2.0"-Standard kompatibel und erzeugt aus dem C++-Quelltext direkt ausführbaren Code, ohne Umwege über C, Assembler oder andere Sprachen. Trotzdem läßt er sich auf ANSI-C-Emulation umschalten, um volle Kompatiblität zu den vorhandenen C-Compilern zu erreichen, und kann optional auch Assembler-Quelltext für die Weiterverarbeitung mit einem fremden Assemblern erzeugen.

Der Compiler wird in gleich zwei Versionen geliefert: Eine um einen Editor aufgebaute, integrierte Entwicklungsumgebung bietet höchsten Komfort, gleichermaßen für das bequeme Erstellen kleiner Programme wie für die Entwicklung umfangreicher Projekte. Eine konventionelle Kommandozeilen-Version erleichtert die Integration in existierende Umgebungen und Tools.

Lieferumfang

Das komplette MaxonC++ Compilersystem besteht aus insgesamt 10 Handbüchern, die je nach Compilerversion geliefert werden:

MaxonC++ Light:

1. MaxonC++ Vorwort
2. MaxonC++ Anwender-Handbuch
3. Edward
4. MaxonC++ Tutorial
5. MaxonC++ Referenz-Handbuch

Die MaxonC++ Normal-Version besteht aus drei weiteren Handbüchern:

6. Maxon MakeApp
7. Ressource-Bibliothek
8. Maxon HotHelp

In der MaxonC++ Developer-Version werden alle Handbücher mitgeliefert!

9. Maxon Source-Level-Debugger
10. MaxonASM (CLI-Version)

Der Compiler und die Zusatzprogramme werden auf mehreren Disketten geliefert. Genaue Angaben zur Installation und zur Diskettenanzahl entnehmen Sie bitte der „Readme"-Datei auf der Installationsdiskette. Da sich nach der Drucklegung noch einige Änderungen, was die Diskettenanzahl und die Programmzusammenstellung angehen, ergeben können, ist dies der beste Weg, um das Handbuch so aktuell wie möglich zu halten.

Upgrades

Allen registrierten Besitzern der MaxonC++ Light und Normalen Compiler-Version bieten wir die Möglichkeit durch Upgrades die MaxonC++-Developer-Version zu erhalten.

Die Upgradekonditionen erfragen Sie bitte bei MAXON Computer.

Registrierung

Ihrem Compilersystem liegt eine Registrierkarte bei, die Sie umgehend an uns zurücksenden sollten. Sie gewährleisten dadurch, daß Sie regelmäßig über Produktneuerungen und Neuerscheinungen informiert werden. Desweiteren steht Ihnen dann unsere tägliche Hotline für Problemlösungen zur Verfügung.

Hotline und Support

Unsere Hotline ist werktags zwischen 15:00 und 17:00 Uhr unter der Rufnummer 06196 / 43780 zu erreichen. Bitte prüfen Sie vor einem Anruf genau den Problemfall:

- Haben Sie die Software richtig installiert?
- Benutzen Sie die richtige Workbench Version (nicht unter 1.3)?
- Laufen irgendwelche Programme im Hintergrund (Screenblanker, Maustools o.ä.)?

Bitte haben Sie Verständnis, daß wir keine Garantie für das Zusammenspiel mit anderer Software geben können. Besonders einige Tools, die in der Startup-Sequence mitgestartet werden, können die Arbeitsfähigkeit sehr stören.

Bei einem Anruf halten Sie bitte Ihre Registriernummer, die Versionsnummer des jeweiligen Softwarepaketes und eine genaue Konfigurationsbeschreibung bereit. Die Registriernummer ist auf den im Handbuch eingehefteten, zurückbleibenden Abschnitt der Registrierkarte aufgedruckt. Die Versionsnummer finden Sie entwender im Info-Dialogfenster des jeweiligen Programmes oder wenn Sie (ab Amiga OS 2.0) den Befehl:

```
Version <Programmname> full
```

in der Shell (CLI) eingeben.

Wenn Sie uns schreiben, benötigen wir zum zügigen Bearbeiten, ebenfalls die beschriebenen Angaben.

Richten Sie Ihr Anschreiben an folgende Adresse:

Maxon Computer GmbH
"MaxonC++-Support"
Postfach 5969
65734 Eschborn

Literaturangaben

Wenn Sie bereits in einer anderen Programmiersprache als C++ programmieren können, ist das Tutorial-Handbuch eine sehr gute Einführung für Sie, um die Syntax und den Umgang mit C++ kennenzulernen. Als Programmieranfänger empfiehlt es sich jedoch, ein Grundeinführungsbuch durchzuarbeiten. Die nachfolgende Liste enthält eine Auswahl an Büchern zu den Themen objektorientierte Programmierung im allgemeinen, Programmierung in C und C++ und zur Programmierung des Amiga Betriebssystems.

Objektorientierte Programmierung allgemein

Objektorientierte Programmierung - Eine Einführung
R. Engel - 338 Seiten - ISBN 3-89090-353-3
Markt&Technik Verlag AG/Haar

C und C++ Literatur

Lernen und Üben: C-Schulung
H. Niegemann und T. Schäfer - 354 Seiten - ISBN 3-89090-218-9
Markt&Technik Verlag AG/Haar

Die Einführung: C
S. Baloui - 379 Seiten - ISBN 3-89090-334-7
Markt&Technik Verlag AG/Haar

Programmieren in C++
E. Huckert - 348 Seiten - ISBN 3-89090-171-9
Markt&Technik Verlag AG/Haar

Clever Programmieren: C++
M. Haun - 352 Seiten - ISBN 3-88322-327-1
IWT Verlag GmbH/Vaterstetten

IWT Kompakt: C-Referenz für alle ANSI-C-Compiler
P. Prinz - 304 Seiten - ISBN 3-88322-340-9
IWT Verlag GmbH/Vaterstetten

C++ lernen und beherrschen: Einführung und Leitfaden
S. B. Lippmann - ca. 500 Seiten - ISBN 3-89319-375-8
Addison-Wesley (Deutschland) GmbH/Bonn

Die C++ Programmiersprache
B. Stroustrup - 380 Seiten - ISBN 3-89319-386-3
Addison-Wesley (Deutschland) GmbH/Bonn

The C++ Programming Language
B. Stroustrup - 576 pages - ISBN 0-201-53992-6
Addison-Wesley (Deutschland) GmbH/Bonn

The Annotated C++ Reference Manual
M. A. Ellis/ B. Stroustrup - 400 pages - ISBN 0-201-51459-1
Addison-Wesley (Deutschland) GmbH/Bonn

Die Kunst der objektorientierten Programmierung - Prinzipien und Konzepte in C++
A. Schäpers - ca. 440 Seiten - ISBN 3-89319-226-3
Addison-Wesley (Deutschland) GmbH/Bonn

C-Programmieren: Ein Kurs zum Selbststudium
G. Krüger - ca. 500 Seiten - ISBN 3-89319-372-3
Addison-Wesley (Deutschland) GmbH/Bonn

Der C-Experte: Programmieren ohne Pannen
A. Koenig - 190 Seiten - ISBN 3-89319-233-6
Addison-Wesley (Deutschland) GmbH/Bonn

Debugging C
R. Ward - 322 Seiten - ISBN 3-925118-90-X
Addison-Wesley (Deutschland) GmbH/Bonn

C Professional Edition
W. Herglotz - ca 450 Seiten - ISBN 3-89360-004-3
BHV Verlags GmbH/4052 Korschenbroich

Das Einsteigerseminar C
W. Herglotz - 224 Seiten - ISBN 3-89360-606-8
BHV Verlags GmbH/4052 Korschenbroich

Das Einsteigerseminar C++
W. Herglotz - 224 Seiten - ISBN 3-89360-622-X
BHV Verlags GmbH/4052 Korschenbroich

C++ für Programmierer
F. Bause/ W. Tölle - 286 Seiten - ISBN 3-528-05110-8
Friedr. Vieweg & Sohn Verlagsgesellschaft mbH/6200 Wiesbaden

Einführung in die Programmiersprache C++
F. Bause/ W. Tölle - 171 Seiten - ISBN 3-528-14689-3
Friedr. Vieweg & Sohn Verlagsgesellschaft mbH/6200 Wiesbaden

Referenzhandbuch Standart C
P.J. Plauger/J. Brodie - 236 Seiten - ISBN 3-528-04741-0
Friedr. Vieweg & Sohn Verlagsgesellschaft mbH/6200 Wiesbaden

C++ Einführung in die objektorientierte Programmierung
S. Prata - 750 Seiten - ISBN 3-89362-701-4
tewi Verlag

Der schnelle Umstieg auf C++
M. Aupperle - 76 Seiten (Chip Spezial) - ISBN 3-8023-1088-8
Vogel Verlag und Druck KG/8700 Würzburg

C für Umsteiger
B. Klöppel - 122 Seiten (Chip Spezial) - ISBN 3-8023-1081-0
Vogel Verlag und Druck KG/8700 Würzburg

Amiga Bücher

Amiga-Programmierrichtlinien
H. Gzella - 272 Seiten - ISBN 3-87791-049-1
Markt&Technik Verlag AG/Haar

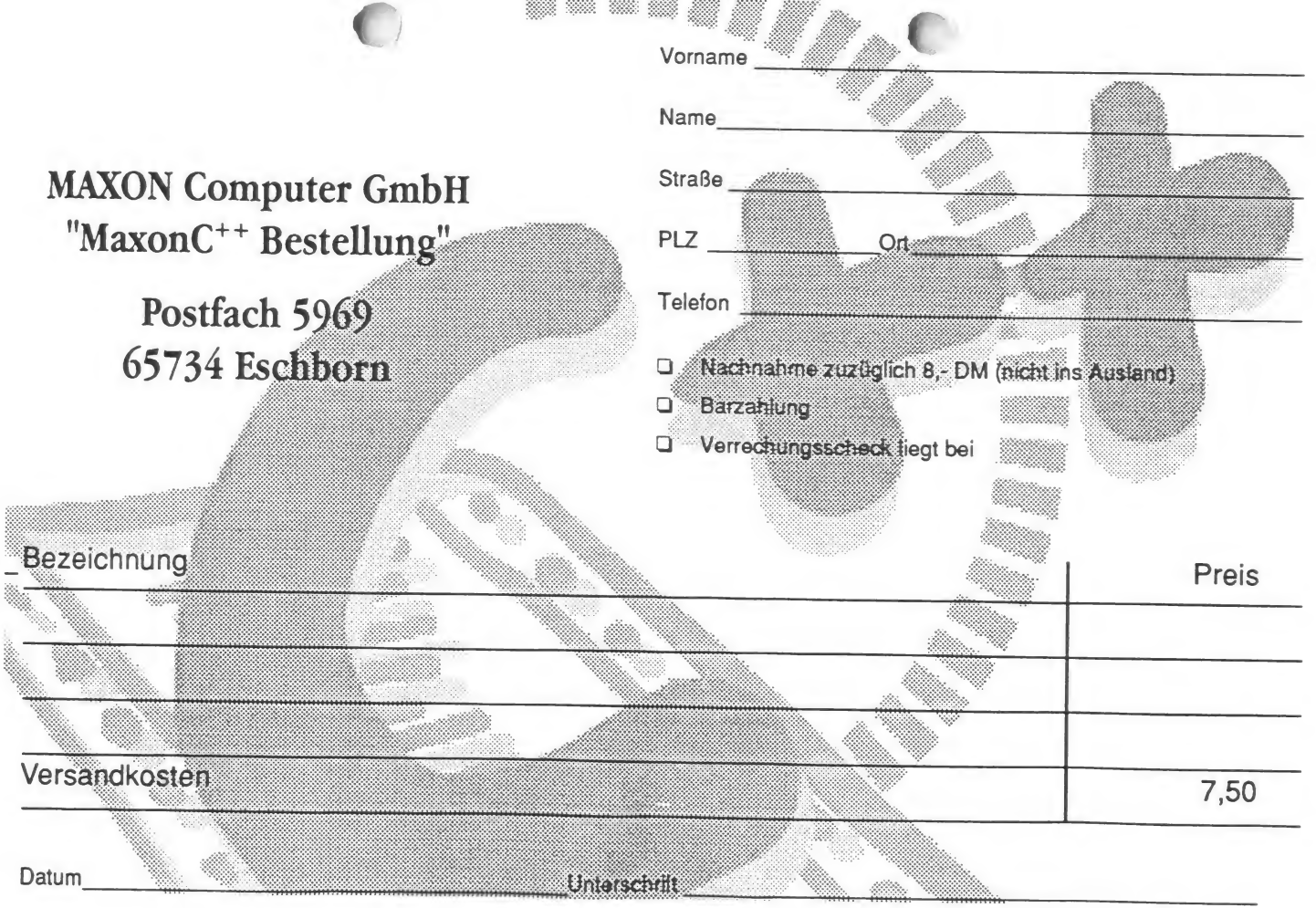
Amiga ROM Kernel Reference Manual: Devices
512 pages - ISBN 0-201-56775-X
Addison-Wesley (Deutschland) GmbH/Bonn

Amiga ROM Kernel Reference Manual: Libraries
960 pages - ISBN 0-201-56774-1
Addison-Wesley (Deutschland) GmbH/Bonn

Amiga ROM Kernel Reference Manual: Includes and Autodocs
1000 pages - ISBN 0-201-56773-3
Addison-Wesley (Deutschland) GmbH/Bonn

Amiga Hardware Reference Manual
512 pages - ISBN 0-201-56776-8
Addison-Wesley (Deutschland) GmbH/Bonn

Amiga User Interface Style Guide
224 pages - ISBN 0-201-57757- 7
Addison-Wesley (Deutschland) GmbH/Bonn



MAXON Computer GmbH
"MaxonC++ Bestellung"

Postfach 5969
65734 Eschborn

Vorname _____

Name _____

Straße _____

PLZ _____ Ort _____

Telefon _____

- ❑ Nachnahme zuzüglich 8,- DM (nicht ins Ausland)
- ❑ Barzahlung
- ❑ Verrechnungsscheck liegt bei

Bezeichnung	Preis
Versandkosten	7,50

Datum _____ Unterschrift _____

Debugger

inkl. 68030 Macro-Assembler

Der Maxon Source-Level-Debugger macht das Aufspühren von Programmfehlern zum Kinderspiel und vervollständigt MaxonC^{++} zum leistungsstarken Compilersystem.

Anhand Ihres Original C- oder C^{++}-Sourcecodes können Sie Ihr Programm Schritt für Schritt ausführen lassen, sich Variablen- oder Strukturinhalte ansehen oder verändern.

Für Profis ist es wichtig, auch den disassemblierten Code zu sehen. Mit MaxonSDB ist dies kein Problem. Der Debugger bietet Ihnen die Möglichkeit, den C/C^{++}-Sourcecode auch gemischt mit dem Disassembler-Listing oder nur das Disassembler-Listing anzuzeigen.

Breakpoints ermöglichen eine gezielte Unterbrechnung des Programmes. Selbstverständlich bietet MaxonSDB auch bedingte Breakpoints.

Zusätzlich erhalten Sie die Kommandozeilenversion des MaxonASM, der problemlos als externer Assembler im Compilersystem genutzt werden kann. Er unterstützt die Befehle der Prozessoren 68000/10/20/30, der FPU 68881/2 sowie der MMU 68851. Macros und Includes sorgen für weiteren Komfort.

Bestellen Sie noch heute den MaxonSDB inklusive 68000/10/20/30 Macro Assembler und 160-seitigem, deutschsprachigem Handbuch zum Preis von nur

249,- DM

Jetzt Neu

MakeAPP

inkl. Ressource-Bibliothek

Vereinfachen Sie sich jetzt die Programmierung der Benutzeroberfläche Ihrer Programme.

Mit MakeAPP erzeugen Sie Dialoge und Menüleisten direkt am Bildschirm. "What you see is what you get" gilt durch MakeAPP auch bei der Erstellung von Programmen.

Der Umgang mit dem komplitzierten Datenwirrwar des Betriebssystems findet dadurch ein Ende.

Ein neuartiger Optimierungsmodus verringert die Speicheranforderung der erzeugten Ressourcen. Bei Gadgets mit gleichen Bordertypen oder Images wird bespielsweise auf ein und denselben Datenbestand zugegriffen. Die Ausgabe als optimierter C-Sourcecode ist ebenfalls möglich.

Die Ressource-Bibliothek ermöglicht Ihnen die einfache Handhabung der erzeugten Ressourcen in Ihren Programmen und stellt zudem noch wichtige Standard-Requester wie File-, Screenmode- und Alert-Requester (wahlweise ASL) auch unter Kickstart 1.3 zur Verfügung.

Bestellen Sie noch heute MakeAPP inklusive 140-seitigem, deutschsprachigem Handbuch zum Preis von nur

129,- DM

Jetzt Neu!

HotHelp

inkl. OS 2.1 Beschreibung

Erfahren Sie alles über die Programmierung Ihres AMIGA. Mit HotHelp wird es Ihnen sehr leicht gemacht!

HotHelp kann jederzeit mit einem einzigen Tastendruck aufgerufen werden und bietet Ihnen die Information, die Sie in dem Moment benötigen.

HotHelp bietet Einsteigern und Profis leichtverständliche Informationen zu allen wichtigen AMIGA-Begriffen und zur Workbench. Eine komplette Aufstellung der Shell- bzw. CLI-Befehle mit ausführlicher Beschreibung, Syntax und Beispielen fehlt genausowenig wie die komplette Übersicht aller Funktionen und Strukturen des ANSI-C-Standards und des AMIGA-Betriebssystems (Libraries & Devices 2.1) mit Angaben über Datentypen, Übergaberegister, Parameter und Library-Offsets.

HotHelp kann auf jedem AMIGA-Modell und unter jeder Betriebssystemversion (>1.2) eingesetzt werden.

HotHelp ist im Kern eine Shared-Library, auf deren Funktionen auch jedes andere Programm zugreifen kann.

HotHelp läßt sich auch als kontextsensitives Hilfesystem für eigene Programme nutzen.

Bestellen Sie noch heute HotHelp inklusive deutschsprachiger Beschreibung der Funktionen zu Amiga-OS 2.1 und 80-seitigem, deutschsprachigem Handbuch zum Preis von nur

89,- DM

Maxon C++

Anwender-Handbuch

MAXON computer

Inhaltsverzeichnis

1. Ein erster Überblick 7

1.1 Die Komponenten des MaxonC++ Systems 7

1.2 Installation 7

1.3 Erste Schritte mit „MaxonC++“ 7

2. Die integrierte Entwicklungsumgebung 9

2.1 Programmstart und Optionen 9

2.2 Compiler und Editor - eine harmonische Beziehungskiste 10

2.3 Das Konzept 12

2.4 Compiler-Funktionen 13
2.4.1 Programme übersetzen 13
2.4.1.1 Das Compiler-Fenster 13
2.4.1.2 Abbruch und Ausstieg 15
2.4.1.3 Auswahl von Fehlermeldungen 16
2.4.1.4 Vom Compiler in den Editor 16
2.4.1.5 „Continue“ und „Run“ 17
2.4.2 Ausführen und Abbrechen 17
2.4.3 übersetzen von Dateien 18
2.4.4 Exe- und Objektdateien 18
2.4.5 Daten und Informationen 19

2.5 Features und Optionen 21
2.5.1 Ein kurzer Überblick 21
2.5.2 Compiler-Optionen 21
2.5.2.1 Include-Dateien 21
2.5.2.2 Der Arbeitsspeicher des Compilers 22
2.5.2.3 Vordefinieren von Symbolen 23
2.5.2.4 Diverse kleine, aber trotzdem wichtige Schalter 23
2.5.2.4.1 C++ 23
2.5.2.4.2 Unterbrechen 24
2.5.2.4.3 IntMult 32 24
2.5.2.4.4 Debug-Datei 24

2.5.2.4.5 Symbol-Hunk 24
2.5.2.4.6 Asm schreiben 25
2.5.2.4.7 Source in Asm 26
2.5.2.4.8 Optimieren 26
2.5.2.5 Optionale Warnungen und Meldungen 26
2.5.3 Das Linker-Dialogfenster 28
2.5.3.1 Datenmodelle 29
2.5.3.2 Linken oder nicht Linken - das ist hier die Frage 30
2.5.3.3 Bibliotheken 31
2.5.3.4 Zusammenfassen von Hunks 31
2.4.3.5 Schreiben von Dateien 32
2.5.4 „MaxonC++"-Systemeinstellungen 32
2.5.4.1 Autosave 32
2.5.4.2 Erzeugen von Icons 33
2.5.4.3 „MaxonC++" - ein polyglottes System 33
2.5.4.4 Protokollieren von Ein- und Ausgaben 33
2.5.4.5 Aufruf eines externen Assemblers 34
2.5.4.6 Argumente aus der imaginären Kommandozeile 34
2.5.4.7 Compiler- und Laufzeit-Fenster 34
2.5.4.8 Stackgröße 35
2.5.5 Aktuelles Verzeichnis 35
2.5.6 Voreinstellungen mit der „Config"-Datei 35

3. MAKE - eine integrierte Projektverwaltung 39

3.1 Das Prinzip 39
3.1.1 Wozu eine Projektverwaltung? 39
3.1.2 Was ist das überhaupt - ein Projekt? 39
3.1.3 Was hat das ganze mit „Make" zu tun? 40

3.2 Die Praxis 41
3.2.1 Erstellen von Projekten 41
3.2.2 Hinzufügen von Dateien, umständliche Methode 41
3.2.3 Weitere Funktionen 43
3.2.4 Externe Übersetzungen 44

3.3 Handgemachte Makedateien 45

4. Der Linker 48

4.1 Prinzipielles 48

4.2 Standard-Funktionen und das „Logfile" 50

4.3 Das Hauptprogramm und seine Argumente 52

4.4 Linker-Bibliotheken 53

4.5 Initialisierungen und das große Aufräumen 54

4.6 Mit und ohne Startup-Code 54

5. Die CLI-Version „CPPC“ 57

5.1 Das Konzept 57

5.2 Optionen 58
5.2.1 Allgemeines 58
5.2.2 Ergebnisse und Ausgaben 59
5.2.2.1 Assembler-Quelltext 59
5.2.2.2 Objektdateien: -c 59
5.2.2.3 Ausführbare Programmdatei: -o 60
5.2.3 Preprozessor 60
5.2.3.1 Definieren von Symbolen: -d 60
5.2.3.2 Includedateien: -i 61
5.2.4 Compiler-Modus: -pc und -pp 61
5.2.5 Codeerzeugung 62
5.2.5.1 Breakpoints setzen: -gb 62
5.2.5.2 Datenmodell: -gd und -gD 62
5.2.5.3 Zusammenfassen von Hunks: -gh 62
5.2.5.4 Ganzzahlige Arithmetik: -gm 62
5.2.5.5 Optimierungen: -go und -gO 63
5.2.5.6 Linken ohne Startup-Code: -gs, -gsi und -gsw 63
5.2.6 Debugger 63
5.2.6.1 Maxon Debugger: -b und -bi 63
5.2.6.2 Symbolischer Debugger: -bs 63
5.2.7 Warnungen und Fehler 63
5.2.7.1 Alternative Fehlerbehandlung: -e und -en 63
5.2.7.2 Optionale Warnungen: -w 64
5.2.8 Linker 65
5.2.8.1 Library-Pfade: -j 65
5.2.8.2 Linker-Libraries: -l 65
5.2.9 Die integrierte Projektverwaltung: -m 65
5.2.10 Arbeitsspeicher: -w 66

5.3 Voreinstellungen mit „CPPOPT“ 66

5.4 Zusammenfassung 67

1. Ein erster Überblick

1.1 Die Komponenten des MaxonC++ Systems

MaxonC++ wird auf mehreren Disketten ohne Kopierschutz geliefert. Die genaue Anzahl und den Inhalt der Disketten erfahren Sie, wenn Sie die „Readme"-Datei auf der ersten Diskette lesen. Da sich nach der Drucklegung dieses Handbuches noch Änderungen ergeben können, wählten wir diesen Weg.

1.2 Installation

Um das System auf Ihrer Festplatte zu installieren, sollten Sie zunächst ein geeignetes Verzeichnis anlegen, beispielsweise mit dem Namen „dh0:MCPP". Sie sollten Datei- oder Verzeichnisnamen mit einem „+" am Ende meiden und das Verzeichnis nicht unbedingt „C++" nennen. Starten Sie jetzt das Installationsprogramm von der ersten Diskette. Alles weitere erfahren Sie aus der „Readme"-Datei.

1.3 Erste Schritte mit „MaxonC++"

Die integrierte Entwicklungsumgebung „MCPP" kann wahlweise von der Workbench oder vom CLI gestartet werden. Dabei besitzt das Programm keine eigene Benutzeroberfläche, sondern lädt den Editor „Edward" nach. Folglich muß dieser Editor korrekt installiert sein. Wird der Edward nicht gefunden, wird „MaxonC++" gleich mit einem entsprechenden Hinweis beendet.

Nach dem Start landen Sie also automatisch im Editor. Hier können Sie Ihre C++-Programme schreiben und bearbeiten. Dem „Edward" ist ein separates Handbuch gewidmet, deshalb soll hier nicht näher auf die Editor-Funktionen eingegangen werden.

Am besten machen Sie sich jetzt ein wenig mit dem Editor vertraut, indem Sie ein kleines Programm schreiben. Sie könnten z. B. folgendes unvermeidliche und traditionsreiche Prgrämmchen abtippen:

```
#include <stream.h>

void main()
{
  cout « "Hello, World!\n";
}
```

Nun haben Sie ein Programm als Quelltext vorliegen und können den Compiler ausprobieren. Zuerst sollten Sie den Text abspeichern, um ihm so einen sinnvollen Namen zu geben, mit dem der Compiler auch etwas anfangen kann. Dann wählen Sie einfach den Menüpunkt „Übersetzen" aus der Menüleiste und warten ab, was geschieht.

Nun öffnet sich ein Fenster, in dem „MaxonC++" einige mehr oder weniger kryptische Meldungen ausgibt. Tritt während der Übersetzung ein Fehler auf, wird er in diesem Fenster gemeldet und der entsprechende Quelltextbereich angezeigt. Sie können dann eine Fehlermeldung anwählen und mit den Symbolen „Edit" und „Close&Edit" direkt die entsprechende Stelle im Editor anspringen.

Wenn das System korrekt installiert wurde und Ihr Programm ebenfalls fehlerlos ist, sollte das Fenster aber ebenso schnell verschwinden, wie es aufgetaucht ist. Nun können Sie die Menü-Funktion „Ausführen" anwählen und das Programm starten. Auch dabei öffnet sich ein Fenster, in dem (wer hätte das gedacht!) „Hello, World!" ausgegeben wird. Außerdem werden Sie freundlich aufgefordert, die „Return"-Taste zu drücken, denn sinnvollerweise schließt „MaxonC++" das Fenster nicht, bevor Sie Gelegenheit hatten, die Ausgabe zu lesen.

Sie können das Programm natürlich noch einmal starten. Wenn Sie aber seit der letzten Übersetzung etwas am Quelltext verändert haben, wird automatisch der Compiler erneut gestartet.

2. Die integrierte Entwicklungsumgebung

2.1 Programmstart und Optionen

Wenn Sie „MCPP“ vom CLI bzw. aus einer Shell starten, können Sie optional einige Parameter angeben. Als erstes können Sie einen Dateinamen angeben, z. B.

```
run MCPP C++:prg/hello.c
```

Sofern die angegebene Datei existiert und lesbar ist, wird sie in den Editor geladen. Da der Edward mehrere Texte verwalten kann, können Sie natürlich auch mehrere Dateinamen angeben.

Wenn Sie eine andere Konfigurationsdatei als „S:MaxonC++.config“ benutzen wollen, können Sie ein zusätzliches Argument mit vorangestelltem „-c“ angeben. Ein Beispiel:

```
run MCPP test.c -cC++:spezial.config
```

Alternativ können Sie auch „c=XXX“ oder „-c=XXX“ setzen. Wichtig ist aber, daß hinter dem „c“ kein Leerraum steht, denn der Parameter „-c“ allein veranlaßt das Programm, keine Konfigurationsdatei zu laden und die jeweiligen Defaultwerte zu verwenden.

MaxonC++ besitzt eine integrierte Projektverwaltung, allgemein bekannt als „Make“. Sie können gleich beim Programmstart eine solche Make-Datei öffnen, und zwar mit der Option „-m“:

```
run MCPP -m plot.makefile
```

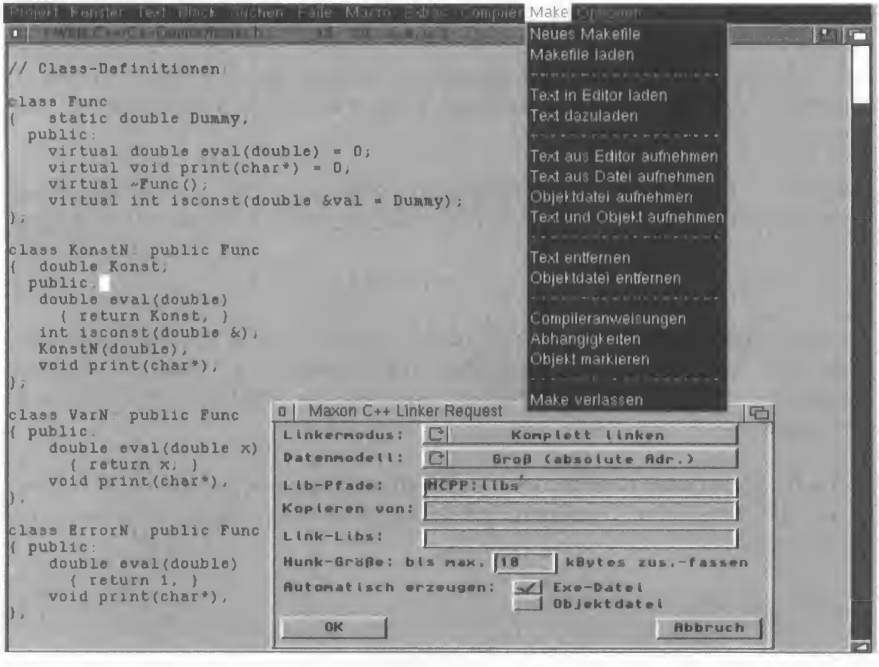
Wie Sie sehen, darf bei dieser Option Leerraum zwischen „-m“ und dem Dateinamen stehen. Auch hier sind wieder die Varianten „m=“ und „-m=“ erlaubt.

Natürlich lassen sich diese Optionen und Argumente in beliebiger Reihenfolge kombinieren, z. B.

```
"MCPP" fclass.c -m plot.makefile -c=df1:s/MaxonC++.config
```

2.2 Compiler und Editor - eine harmonische Beziehungskiste

Wie oben schon angedeutet, ist der Editor „Edward" ein eigenständiges Programm, das auch ohne den Compiler benutzt werden kann. „MaxonC++" lädt ihn einfach nach und startet ihn als zweiten Prozess, läßt aber keinen Zweifel daran aufkommen, daß er der Boß ist.



Weniger blumig ausgedrückt, richtet „MaxonC++" einen Port ein, über den sich Editor und Compiler Nachrichten zuschicken. Wird im Edward vom Benutzer eine Compiler-Funktion aufgerufen, etwa durch eines der Pull-Down-Menüs oder eine entsprechend belegte Tastenkombination, läßt Edward dem Compiler eine entsprechende Nachricht zukommen.

Ich quäle Sie mit diesen implementatorischen Details, damit Sie verstehen können, wie Sie das „MaxonC++"-System selbst konfigurieren können. Der Schlüssel dazu ist die „.DEF"-Datei des Edward. Hier werden alle Menüs und Tastenbelegungen des Editors abgelegt, und hier muß irgendwie die Benutzerschnittstelle des Compilers integriert werden, auch wenn der Editor nicht die geringste Anhnug hat, was ein Compiler ist und wozu er gut ist.

Deshalb hat der Edward eine Spezial-Funktion namens „call", die als Argument wiederum eine Compiler-Funktion erwartet. Beispielsweise gibt es eine „MaxonC++"-Anweisung namens „compile". Wird nun vom Benutzer, also Ihnen, im Edward ein Menü ausgewählt, das mit der Funktion „call compile" belegt ist, schickt Edward das „compile" als Nachricht an den „MaxonC++" und kümmert sich dann nicht weiter drum. „MaxonC++" dagegen erhält diese Anweisung, fordert vom Edward noch ein paar zusätzliche Daten an und beginnt dann freudig mit der Übersetzung. Natürlich kann man nicht nur ein Menü, sondern auch eine Taste oder Tastenkombination mit diesen „call"-Anweisungen belegen.

Auf diese Weise sind der Compiler und der Editor vollkommen voneinander abgekoppelt, und Edward braucht nichts über „MaxonC++" zu wissen. Nun kommt es natürlich vor, daß Sie den Edward einmal „einfach so" ohne Compiler benutzen wollen. Deshalb läßt Edward einen kleinen „Preprozessor" auf das „.DEF"-File los und filtert je nach Modus Teile aus dem Text aus - ähnlich wie ein C-Preprozessor bei „#ifdef"-Zeilen. Auch dieses Feature ist in Ihrem Edward-Handbuch beschrieben. „MaxonC++" teilt dem Edward mit, daß ein Symbol namens „COMPILER" definiert sein soll. Nun können Sie Dinge wie „#ifdef COMPILER" in Ihr Defs-File schreiben, um Definitionen je nach Modus ein- oder auszublenden. Insbesondere werden Sie sich um sämtliche Menü-Definitionen des Compilers eine „#ifdef COMPILER ... #endif"-Klammer legen wollen, wie es auch im mitgelieferten Standard-File der Fall ist.

Langer Rede, schwacher Sinn: Wenn ich im folgenden von irgendwelchen Menüs schwafele, unterschlage ich dabei jedesmal Hinweise wie „...oder wie auch immer Sie diese Funktion konfiguriert haben". Ich meine damit jeweils immer die Funktionen, die in der Standard-Konfiguration auf dem entsprechenden Menü liegen.

Es gibt übrigens standardmäßig folgende Menüs, mit den nachstehend in voller Länge und der jeweiligen Kurzform angegebenen Compiler-Funktionsnamen:

- Compiler
  - Übersetzen — compile — c
  - Ausführen — run — r
  - Abbrechen — break — b
  - Compile Datei — compfile — f
  - Exe-Datei — exefile — x
  - Objektdatei — objectfile — o
  - Information — info — i

- Make
  - Neues Makefile — newmk — nm
  - Makefile laden — loadmk — lm
  - Text in Editor laden — opentext — ot
  - Text aus Editor aufnehmen — addedittx — ae

- Text aus Datei aufnehmen	addfiletx	af
- Objektdatei aufnehmen	addobj	ao
- Text und Objekt aufnehmen	addtxobj	ato
- Text entfernen	remtx	rt
- Objektdatei entfernen	remobj	ro
- Compileranweisungen	mkinstr	in
- Abhängigkeiten	mkdep	dp
- Objekt markieren	touch	to
- Make verlassen	leavemk	qm

- Optionen

- System	opsys	os
- Compiler	opcomp	oc
- Linker	oplink	ol
- Verzeichnis	chdir	d
- Optionen sichern	saveconf	as

2.3 Das Konzept

Auf die Gefahr hin, daß Sie mich dafür steinigen, werde ich trotzdem noch ein paar grundsätzliche Worte loslassen. Prinzipiell kann „MaxonC++" in zwei verschiedenen Modi arbeiten: Im „normalen" Modus, in dem man vor allen kleinere Programme (mit nur einem Modul) schreibt, und im „Make"-Modus. Da der Projekverwaltung „Make" ein eigenes Kapitel gewidmet ist, befassen wir uns zunächst nur mit dem Standard-Modus, der natürlich auch aktiv ist, wenn Sie das „MaxonC++"-System starten.

Auch wenn Edward beliebig viele Texte gleichzeitig bearbeiten kann, verwaltet „MaxonC++" immer nur ein Programm. Das hat zum einen Gründe, die mit dem Speicherplatz zusammenhängen, und resultiert zum anderen aus der Beobachtung, daß das in der Praxis nicht besonders relevant ist, weil man normalerweise zwar durchaus mit mehreren Texten, aber an nur einem Programm arbeitet.

Die Funktion „compile" übersetzt stets den Text im gerade aktiven Fenster. Bei „run" wird immer zuerst folgendes geprüft:

- Liegt überhaupt ein übersetztes Programm vor?
- Befindet sich der Quelltext dieses Programms im aktiven Editor-Fenster?
- Blieb er seit der letzten Übersetzung unverändert?

Falls eine der drei Fragen mit „Nein" beantwortet werden muß, wird die „compile"-Funktion aufgerufen und damit der Text im aktiven Fenster übersetzt.

So, das sollte gesagt werden, damit Sie beim Rumprobieren mit „MaxonC++" (was Sie wahrscheinlich schon längst getan haben) nicht in vermeidbare Verwirrung stürzen (was dabei hoffentlich nicht geschehen ist). Und nun folgt endlich eine ausführliche Beschreibung der Funktionen von „MaxonC++":

2.4 Compiler-Funktionen

2.4.1 Programme übersetzen

2.4.1.1 Das Compiler-Fenster

Am besten fangen wir mit der Funktion an, wegen der Sie „MaxonC++" überhaupt gekauft haben, nämlich dem Übersetzen von Programmen.

Wenn Sie diese Funktion auswählen, wird - je nach Modus - entweder das Programm im gerade aktiven Editor-Fenster übersetzt, oder das integrierte „Make" entscheidet selbst, welche Objektdateien neu übersetzt werden sollen. In jedem Fall öffnet sich aber ein Fenster, das in fünf Bereiche aufgeteilt ist.

Am oberen Rand befindet sich die Statuszeile. Hier wird stets links der Name der gerade überstzten Datei angezeigt, während „MaxonC++" rechts sagt, was gerade getan wird bzw. zuletzt getan wurde.

Am besten probieren Sie es einmal aus: Laden Sie einen C-Quelltext (z. B. eines der Beispielprogramme) in den Editor und wählen Sie den Menüpunkt „Übersetzen" aus. Wenn alles klappt, wird oben rechts in schneller Folge (bei kurzen Programmen wohl zu schnell zum Mitlesen) das folgende Ausgegeben:

- „Compiling": Der Compiler liest und analysiert das Programm.

- „No Errors" bzw. „X Errors found": Nachdem der Quelltext gelesen und auf seine Korrektheit geprüft wurde, erzeugt der Compiler nun den Code dazu.

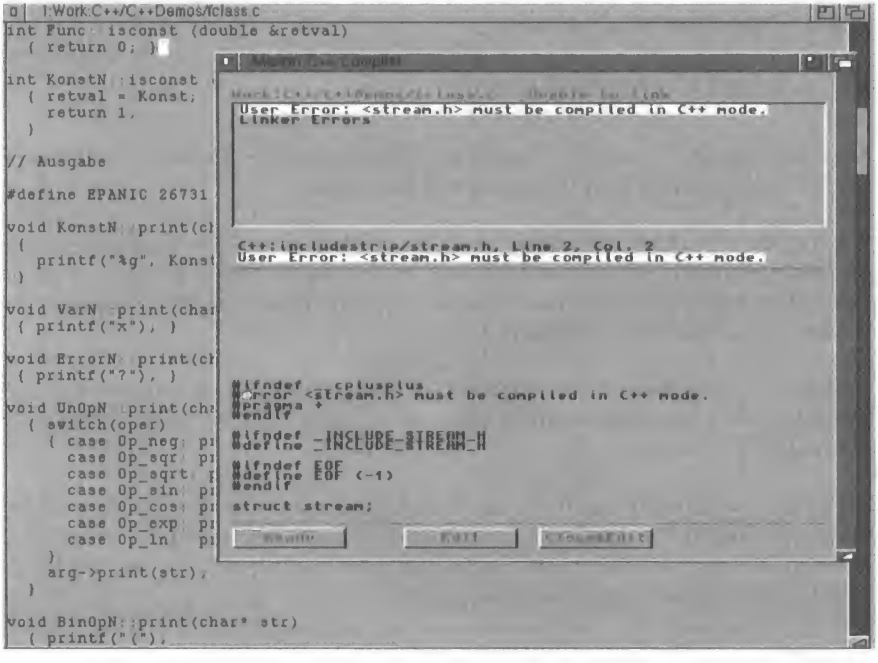
- „Linking": Der Linker bindet das erzeugte Programm mit den nötigen Bibliotheksfunktionen zusammen.

- „Ready": Das Programm ist übersetzt und ausführbar. Wurden keine Fehler oder Warnungen ausgegeben, verschwindet das Fenster bei dieser Meldung automatisch.

Natürlich kann dabei beliebig viel schief gehen, beispielsweise könnte das Programm fehlerhaft sein. Fehlermeldungen und Warnungen aller Art werden in dem Rahmen unterhalb der Statuszeile ausgegeben. „MaxonC++" gibt immer höchstens so viele Fehler

aus, wie in diesen Rahmen passen (je nach Windowgröße zwischen zwei und zehn Stück) und wartet auf eine Reaktion des Benutzers, falls die Höchstzahl erreicht wurde.

Eine solche einzeilige Fehlermeldung allein ist meist natürlich nicht aussagekräftig genug. Deshalb kann sich der Programmierer jeweils zu einer Meldung ausführlichere Informationen anzeigen lassen.



Der Bereich zwischen dem Fehlermeldungs-Rahmen und der unteren Symbolleiste wird im Fehlerfalle durch waagerechte Leisten zweigeteilt. In den beiden oberen Zeilen stehen Quelltextdatei, Zeilen- und Spaltennummer des gerade angewählten Fehlers und noch einmal eine Wiederholung der schon im Rahmen ausgegebenen Fehlermeldung. Unterhalb der horizontalen Linie sehen Sie einen Ausschnitt aus dem Quelltext mit einem Pseudo-Cursor an der fraglichen Position.

Wenn allerdings der Linker Fehler meldet, steht anstelle der Sourceposition (was weiß ein Linker schon von Quelltexten!) schlicht „Linker Error“, und dort, wo sonst der

Quelltext-Ausschnitt dargestellt wird, steht eine eventuell verkürzte Liste der beim Zusammenbinden aufgetretenen Fehler, also hauptsächlich Namen von undefinierten oder gleich mehrfach definierten Linker-Symbolen.

Weniger häufig treten beim Übersetzen unter dem Titel „Fatal Error" Fehler bei der Codegenerierung oder beim Assemblieren auf. Meist sind diese auf Speichermangel zurückzuführen, und dementsprechend heißen sie „Out of Memory", „Workspace overflow".

Last not least gibt es ganz unten im Compiler-Fenster noch eine Symbolleiste. Zunächst finden Sie hier nur ein Symbol mit der Aufschrift „Break". Bei Bedarf werden aber weitere Symbole hinzugefügt.

2.4.1.2 Abbruch und Ausstieg

Im Gegensatz zu einfachen Kommandozeilen-Compilern besitzt das „MaxonC++"-Fenster eine Vielzahl von Knöpfen und Funktionen, an denen der C++-Programmierer herumspielen kann. Alle Symbol-Funktionen sind, um den Bedürfnissen der „echten Programmierer" entgegenzukommen, auch durch Tasten oder Tastenkombinationen bedienbar.

Beim Start des Compilers können Sie im Compiler-Fenster zunächst nur zwei Dinge tun: Das Fenster schließen oder die Übersetzung abbrechen.

Mit einem Klick auf das Schließfeld wird die Übersetzung abgebrochen und das Fenster geschlossen. Die Taste „ESC" hat übrigens die gleiche Wirkung.

Außerdem gibt es unten rechts das bereits erwähnte „Break"-Symbol. Man benutzt es, wenn man den Compiler zwar abbrechen, aber noch irgendwelche im Fenster ausgegebenen Meldungen in Ruhe lesen möchte. Nach einem Klick auf „Break", aber auch dann, wenn der Compiler fertig ist und Ihnen noch Zeit zur Lektüre der Fehlermeldungen geben möchte, verwandelt sich die Symbol-Beschriftung in „Ready", was in diesem Fall keineswegs unbedingt heißt, daß der Compiler fertig ist (das wird ja durch das „Ready" oben rechts in der Statuszeile angezeigt), sondern lediglich Sie dazu ermuntern soll, zu bestätigen, daß Sie fertig sind. Nach einem Klick auf das „Ready"-Symbol wird ebenfalls das Fenster geschlossen.

Auch für das „Break"- bzw. „Ready"-Symbol müssen Sie natürlich nicht zur Maus greifen. Es reicht, wenn Sie hier die kanonische Abbruch-Tastenkombination, nämlich Ctrl-C, drücken.

2.4.1.3 Auswahl von Fehlermeldungen

Laut Murphy geht bei so ziemlich jedem Compiler-Lauf etwas schief, und meist sogar etwas mehr als nur „etwas". Deshalb bietet das Compiler-Fenster Platz für bis zu zehn Fehlermeldungen auf einmal.

Eine dieser jeweils einzeiligen Meldungen ist immer invertiert und somit hervorgehoben, und zu genau dieser Meldung werden unterhalb des Rahmens zusätzliche Informationen angezeigt. Mit einem Mausklick auf eine Fehlerzeile macht man diese zur gerade privilegierten, invertierten Meldung.

Das geht aber auch über die Tastatur: Mit der Leertaste durchläuft man die Fehlermeldungen von oben nach unten, und mit „Backspace" in die andere Richtung.

2.4.1.4 Vom Compiler in den Editor

Der besondere Witz an einer integrierten Umgebung ist ja das perfekte Zusammenspiel von Compiler und Editor, insbesondere das automatische Anspringen von Fehlerstellen.

Das funktioniert in „MaxonC++" denkbar einfach: Sobald eine Fehlermeldung im Compiler-Fenster ausgegeben wird, erscheinen zwei Symbole mit der Aufschrift „Edit" bzw. „Close&Edit". Nun wählen Sie wie oben beschrieben Ihre Wunsch-Fehlermeldung aus (das geht übrigens auch, während der Compiler noch arbeitet) und klicken „Edit" an. Prompt lädt Edward die zugehörige Quelltextdatei (falls der Text nicht ohnehin schon im Editor vorliegt) und setzt den Cursor an die bemängelte Position. Nun können Sie den Fehler berichtigen - vorausgesetzt natürlich, der Compiler läuft nicht noch und hat ein „Lock" auf diesen Text eingerichtet. Das Compiler-Fenster wird dabei nicht geschlossen, so daß Sie noch weitere Fehlerstellen anspringen können.

Die Integration von Compiler und Editor geht allerdings nicht so weit, daß im Compiler-Fenster Quelltext-Veränderungen beachtet würden. Wenn Sie z. B. zuerst einen Fehler berichtigen und dabei einige Zeilen einfügen oder löschen müssen, merkt „MaxonC++" nichts davon. Sollten Sie also anschließend eine weiter unten im Quelltext gelegene Fehlerstelle anspringen lassen, landen Sie gnadenlos in der Zeile mit der Nummer, in der der Fehler lag, als er bemerkt wurde - und durch die inzwischen vorgenommenen Änderungen stimmt diese Zeilennummer vielleicht nicht mehr, und Sie landen einige Zeilen zu hoch oder zu tief. Das dürfte aber in der Praxis kaum ein Problem darstellen, denn im Compiler-Fenster wird immer noch der ursprüngliche Quelltext mit dem Cursor an der richtigen Stelle angezeigt, und durch einen kurzen Textvergleich zwischen Compiler- und Editorfenster können Sie schnell die wirkliche Fehlerposition „zu Fuß" ansteuern. Außerdem ändert man in der Praxis bei Compilerfehlern meist sowieso nur die fragliche Zeile, ohne großartiges Löschen und Einfügen von Zeilen. Das Symbol „Edit" ist mit dem Hotkey „E" belegt.

Manchmal tritt während der Übersetzung nur ein einziger Fehler auf, der vielleicht noch den einen oder anderen Folge-Fehler nach sich zieht, aber trotzdem nur eine einzige Änderung am Quelltext notwendig macht. Deshalb gibt es auch noch das Symbol „Close Edit". Hier wird im Editor die gerade angewählte Fehlerposition angesprungen (genau wie beim „Edit"-Gagdet) und gleichzeitig das Compiler-Fenster geschlossen - wobei natürlich auch der Compiler abgebrochen wird, sollte er noch am arbeiten sein.

Alternativ können Sie auch die „Return"-Taste drücken oder mit der Maus einen Doppelklick auf die Fehlermeldung machen.

2.4.1.5 „Continue" und „Run"

Bei passender Gelegenheit taucht in der unteren rechten Ecke des Compiler-Fensters ein Symbol auf, das entweder mit „Continue" oder mit „Run" beschriftet ist und dementsprechend auch mit dem Hotkey „C" oder „R" belegt ist.

Treten beim Übersetzen bzw. Linken mehr Fehler auf, als im Rahmen dargestellt werden können, wartet der Compiler erst einmal ab, und es erscheint das „Continue"-Symbol. Nach einem Klick darauf vergißt „MaxonC++" die bisher gemeldeten Fehler bis auf den letzten und macht munter weiter. Meist ist es aber nicht unbedingt sinnvoll, nach mehr als zehn Fehlern noch weiter zu übersetzen, denn normalerweise kommt dabei nicht mehr viel sinnvolles ‚raus. Deshalb wird man in der Regel das „Continue"-Symbol meiden und lieber die Fehler korrigieren.

Wenn „MaxonC++" bei der Übersetzung Fehler oder Warnungen ausgegeben hat, aber trotzdem ein lauffähiges Programm erzeugen und linken konnte, taucht unten rechts das „Run"-Symbol auf. Mit einem beherzten Klick darauf können Sie das Programm starten - aber vorher sollten Sie sich genau überlegen, ob die gemeldeten Fehler beim Starten des Programms nicht zu einem Programmabsturz oder ähnlich unangenehmen Effekten führen können!

2.4.2 Ausführen und Abbrechen

Die zweite wichtige Funktion aus dem „Compiler"-Menü heißt „Ausführen" und tut auch genau das: Ein C++-Programm wird gestartet.

Vorher prüft „MaxonC++" allerdings noch, ob nicht vielleicht erst noch der Compiler konsultiert werden muß, und startet selbigen automatisch.

Anschließend wird ein Text-Fenster für die Ein- und Ausgaben des Programms geöffnet. Falls das Programm tatsächlich etwas ausgegeben hat, erscheint nach erfolgreicher Programmausführung die Aufforderung „Press Return“ - Sie sollen also die „Return“-Taste drücken und so bestätigen, daß Sie die Ausgaben des Programms auch gelesen haben.

Während das Programm läuft, sind verschiedene Funktionen des Systems gesperrt, z.B. kann man es nicht ein zweites Mal starten. Man kann den Compiler nicht laufen lassen, weil man dem laufenden Programm ja dadurch den Code unter den Füßen wegziehen bzw. es überschreiben würde. Das ist sicher sinvoll und stört auch nicht weiter - es sei denn, das Programm ist irgendwo hängengeblieben und läßt sich nicht normal beenden.

Gegen „normale” Endlosschleifen hilft die Compileroption „Unterbrechen”. Schwieriger wird es dagegen, wenn der Prozeß abgestürzt ist, auf ein Ereignis wartet, daß nie eintreten kann oder die erwähnte Unterbrechnungs-Option einfach nicht benutzt wurde. Deshalb (und nur für solche Fälle!) gibt es im Compiler-Menü die Funktion „Abbrechen”, die den gestarteten Prozeß schlicht abtötet und dann das Ausgabefenster schließt. Auf den ersten Blick eine praktische Lösung, aber leider gibt es auf dem Amiga (im Gegensatz z.B. zu UNIX) keine saubere Möglichkeit, einen Prozeß aus dem System zu entfernen. Insbesondere werden dabei keine Resourcen (reservierter Speicher, geöffnete Dateien und Fenster etc.) freigegeben. Sie müssen davon ausgehen, daß Sie auf diese Weise freien Speicher verlieren, wenn Sie auf diese brutale Art abbrechen. Aber das ist im Zweifel immer noch angenehmer, als das „MaxonC++”-System neu zu starten oder gar den Rechner zu booten.

2.4.3 übersetzen von Dateien

Als kleines Schmalkerl gibt es auch noch die Menü-Funktion „Compile Datei“, mit der Sie einen Quelltext übersetzen können, ohne ihn in den Editor laden zu müssen. Mit dem Dateiauswahlfenster werden Sie aufgefordert, die Textdatei auszuwählen. Im Falle eines Fehlers stehen Ihnen alle Möglichkeiten des Fehlerstellenansprungs zur Verfügung. Wenn das Programm übersetzt und gelinkt werden konnte, können Sie es ganz wie ein aus dem Editor compiliertes Programm ausführen oder sich Objekt- bzw. Exe-Dateien erzeugen lassen.

2.4.4 Exe- und Objektdateien

Ungeachtet allen damit verbundenen Komforts werden Sie Ihre Programme wohl nicht immer nur aus der integrierten Umgebung starten wollen. Deshalb gibt es im „Compiler“-Menü zwei Funktionen, die das übersetzte Programm in Dateien schreiben:

- „Exe-Datei" erzeugt eine ausführbare Programmdatei, vorausgesetzt natürlich, ein Programm ist korrekt übersetzt und gelinkt worden.

- „Objekt-Datei" schreibt zur zuletzt compilierten Übersetzungseinheit eine Objektdatei, aus der man später z. B. mit dem „CPPC" ein ausführbares Programm erzeugen kann. Im „Make"-Modus ist diese Funktion nicht unbedingt sinnvoll, denn dort werden diese Objektdateien ja beim Compilieren automatisch erzeugt.

2.4.5 Daten und Informationen

Mit dem Menüpunkt „Information" öffnet man ein Fenster, in dem Informationen über den Status von „MaxonC++" angezeigt und ständig aktualisiert werden. Man schließt das Info-Fenster wieder, indem man das Schließsymbol anklickt oder im aktivierten Info-Fenster eine beliebige Taste drückt.

Welche mehr oder weniger hilfreichen Informationen angezeigt werden, hängt von der aktuellen Windowgröße ab. Bei der normalen Größe, wie Sie sie vorfinden, wenn Sie das Fenster öffnen, sehen Sie einen repräsentativen „MaxonC++"-Schriftzug, eine kurze Information über Urheberschaft und Copyright und einige Daten, auf die im folgenden eingegangen werden soll. Wenn Sie das Info-Fenster noch etwas vergrößern, finden Sie etwas ausführlichere Angaben über „MaxonC++" und Maxon, und wenn Sie es hinreichend verkleinern (aber natürlich nicht so klein machen, daß nichts mehr hineinpaßt), stehen dort nur noch die geheimnisvollen Kürzel und Daten.

Ganz allgemein gesagt, stehen dort (falls da überhaut etwas steht, nämlich nur dann, wenn ein compiliertes Programm vorliegt) jeweils von „MaxonC++" Speicherbereiche mit ihren Anfangs- und Endadressen, ihrer Größe und natürlich ihrer Bedeutung.

Sie können dort folgende Daten finden:

Exe Gesamtgröße des ausführbaren Programms, so weit es im FAST-Mem liegt.

Wenn es einen Hunk namens Chip gibt, umfaßt dieser den zusätzlichen Speicherplatz im CHIP-RAM.

Free Der Teil des Arbeitsspeichers („Workspace"), der nicht mit dem Executable ausgefüllt ist.

Code Codehunk des soeben übersetzten Moduls

Data Hunk für initialisierte Daten des Moduls

Bss Speicherbereich für nicht initialisierte Daten

Dt.c Chip-RAM-Daten

SDta Datenhunk für „Small Data". Genau wie „Code", „Data" und „Bss" bezieht sich auch diese Angabe nur auf die jeweils zuletzt compilierte Übersetzungseinheit und ist deshalb im „Make"-Modus nicht so besonders erbaulich.

Natürlich werden nicht immer alle diese Daten ausgegeben, sondern nur die, welche gerade real existieren. Um die Verwirrung komplett zu machen, wird nach erfolgreichem Linken auch noch die „Maximum code size" ausgegeben, also noch ein Maß für den Umfang des erzeugten Codes, aber auch das ist noch nicht genau die spätere Größe einer ausführbaren Programmdatei. Das Ganze hängt folgendermaßen zusammen:

Ein ausführbares Programm besteht aus Daten, die im Chip-Memory liegen müssen (insbesondere Grafikdaten) und solchen, die in einen beliebigen Speicherbereich geladen werden dürfen, vorzugsweise natürlich ins Fast-RAM. Diese beiden Datenbestände werden durch die Angaben „Exe" und „Chip" repräsentiert und ihre Summe ist immer genau die „Maximum code size". Das Wort „maximum" können Sie dabei vorerst getrost vergessen, denn in der vorliegenden Version beherrscht MaxonC++ noch keine Overlays, und der Speicherplatzbedarf ist deshalb eine konstante Größe. Diese Angabe bezieht sich aber auf die Anzahl der Bytes, die das Programm belegt, wenn es in den Speicher geladen wurde, wogegen bei der ausführbaren Programmdatei noch ein gewisser Overhead dazukommt.

Ein zusammengelinktes Programm ist aber kein amorpher Klumpen von Bytes, sondern durch „Hunks" strukturiert. Jede Übersetzungseinheit (C-typische Bezeichnung für „Modul") eines Programms besteht aus wenigstens einem Hunk, nämlich typischerweise einem Code-Hunk für den ausführbaren Code, eventuell je einem Data- oder Bss-Hunk für die initialisierten bzw. uninitialisierten statischen Daten oder alternativ einem gemeinsamen, Adressregister-relativ adressierten Datenhunk (beim „Small Data"-Modell) sowie manchmal einem zusätzlichen Hunk für Daten, die in das Chip-Memory gehören.

MaxonC++ gibt auch die Größe dieser Hunks (soweit vorhanden) aus, aber nur die Hunks des zuletzt übersetzten Moduls. Hinzu kommen stets noch einige Hunks für die Bibliotheken und bei Projekten, die aus mehreren Übersetzungseinheiten bestehen, natürlich die Hunks der übrigen Module. Deshalb ist die Summe der ausgegebenen Hunks nicht identisch mit der Gesamt-Codegröße des Programms.

2.5 Features und Optionen

2.5.1 Ein kurzer Überblick

Für viele Programmierer ist das Schönste an einem Compiler, daß man so viele Optionen und Switches hat, an denen man herumspielen kann. Bei „MaxonC++“ nimmt man sämtliche dieser Einstellungen über Dialogfenster vor, deren es genau drei gibt: Eines („System“) für allgemeine Einstellungen, die in irgendeiner Weise mit dem „MaxonC++“-System zu tun haben, ein weiteres für die unvermeidlichen Compiler-Optionen und schließlich einen eigenes Dialogfenster für die Linker-Steuerung.

Wie Sie sich vielleicht denken können, beherrschen „MaxonC++“ und die Kommandozeilen-Version „CPPC“ praktisch dieselben Einstellungen und Optionen, nur die Bedienung ist eben unterschiedlich. Deshalb sollten Sie sich dieses Kapitel auch dann durchlesen, wenn Sie ausschließlich den „CPPC“ benutzen wollen, denn bei der Beschreibung der Optionen im „CPPC“-Kapitel wird hauptsächlich das „Wie“ der Einstellungen beschrieben, während nur hier ausführlich erläutert wird, was diese Optionen genau bedeuten.

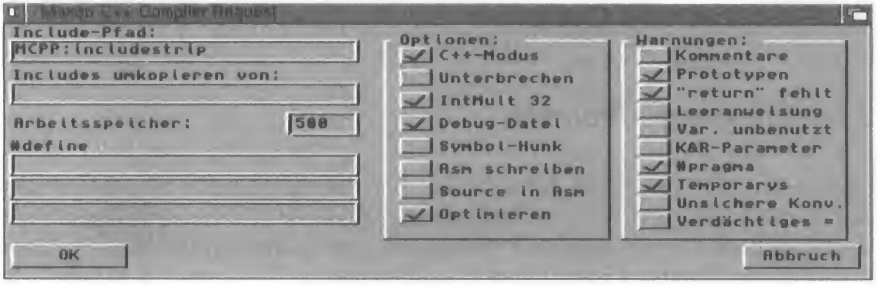
Zurück zum „MaxonC++“: Wenn Sie wollen, können Sie alle drei Dialogfenster gleichzeitig öffnen. In jedem der Fenster finden Sie am unteren Rand die obligatorischen Symbole „OK“ und „Cancel“, mit denen Sie die vorgenommenen Einstellungen bestätigen bzw. sie wieder verwerfen können.

2.5.2 Compiler-Optionen

2.5.2.1 Include-Dateien

Wie eigentlich alle modernen Programmiersprachen, sind C und C++ ziemlich „dumm“: Es gibt für den Compiler keinerlei vordefinierten Funktionen (während z. B. in Pascal „WriteLn“ oder in BASIC „print“ stets zur Verfügung stehen). Vielmehr stehen Definitionen für die Standard-Funktionen in diversen Includedateien, die man explizit einbinden muß.

Vielleicht wissen Sie schon, daß es dafür zwei verschiedene Möglichkeiten gibt: Schließt man den Dateinamen „so“ in Anführungszeichen ein, sucht der Compiler (oder der Preprozessor, um genauer zu sein) vom aktuellen Verzeichnis aus, während die <so> in spitze Klammern gesetzten Dateinamen Standard-Includedateien bezeichnen und in einem ganz bestimmten Verzeichnis gesucht werden. Von genau diesem Verzeichnis, und wie man es festlegt, soll hier die Rede sein.



Im „Compiler"-Dialogfenster legen Sie mit den beiden Texteingabefelder oben links eben dieses Verzeichnis fest. Wenn Sie nur das obere Symbol benutzen, dürfte die Funktionsweise klar sein: Der Compiler lädt die Include-Dateien aus dem dort angegebenen Verzeichnis. Als kleines Bonbon vor allem für Diskettenbenutzer gibt es aber auch noch die Möglichkeit, die benutzten Dateien auf einen schnelleren Datenträger umkopieren zu lassen: Wenn Sie einen doppelten Suchpfad angeben, sucht MaxonC++ zuerst unter dem zuerst genannten Pfad, der z. B. in der RAM-Disk liegen könnte. Findet er die Datei dort nicht, versucht er es über den zweiten Pfad, z. B. auf einer Diskette, und kopiert die Datei in das erste Verzeichnis um.
Beim nächsten Übersetzungsvorgang liegt die Datei dann auf dem schnellen Medium vor. Übrigens legt MaxonC++ vor dem Umkopieren bei Bedarf selbstständig die benötigten Unterverzeichnisse an.

Vor einer kleinen Heimtücke des Systems sei noch gewarnt: Achten Sie bitte darauf, daß Sie weder an den Anfang noch an das Ende der Dateinamen versehentlich irgendwelche Leerzeichen anhängen! „MaxonC++" versucht sonst gänzlich ungerührt, auf die angegebenen Pfade einschließlich der überflüssigen Spaces zuzugreifen, und da wird er naturgemäß nicht allzu viel finden.

2.5.2.2 Der Arbeitsspeicher des Compilers

Hin und wieder braucht der MaxonC++ Compiler einen zusammenhängenden Speicherbereich, nämlich bei Codegenerierung und Assemblierung. Außerdem legt er nach erfolgreichem Linken in diesem Speicherblock eine lauffähige Kopie des erzeugten Programms ab, und sollte dabei noch genug übrig bleiben, benutzt er den Speicher für den Stack, wenn das Programm aus der „MaxonC++"-Umgebung gestartet wird.

Was Sie damit zu schaffen haben? Eigentlich nichts, aber leider weiß „MaxonC++" nicht im Voraus, wieviel Speicher er dafür brauchen wird. Der Compiler kann später höchstens einen Überlauf feststellen, aber vergrößern kann er diesen „Workspace" genannten Speicherblock dann nicht mehr, weshalb er mit einer Meldung wie „Workspace Overflow" aussteigt.

Deshalb müssen Sie leider persönlich festlegen, wie groß dieser Workspace sein soll, und zwar mit den entsprechendem Integer-Symbol im Compiler-Dialogfenster. Der Defaultwert von 80 kByte dürfte übrigens für die allermeisten Zwecke ausreichen, und wenn Sie mit RAM etwas knapp sind, können Sie ruhig versuchen, die Workspace-Größe weiter zu verringern. Sollte der Compiler dagegen einmal „Workspace Overflow" melden, setzen Sie die Größe einfach etwas herauf und versuchen es noch einmal.

2.5.2.3 Vordefinieren von Symbolen

Mit den drei Symbolen, die mit „#define" überschrieben sind, können Sie bis zu drei Preprozessor-Symbole vordefinieren. Jeder Eintrag in einem dieser Texteingabefelder wirkt dann so, als stünde er mit einem vorangestellten „#define" am Anfang jedes übersetzten Quelltexts. Dieses Feature entspricht also der „-d"-Option von „CPPC".

Ein Beispiel: Sie haben als vorsichtiger Programmierer die Includedatei „<assert.h>" eingebunden und diverse Sicherheitsabfragen in Ihr Programm gesetzt - (übrigens ein löbliches Unterfangen). Um diese Abfragen automatisch aus dem Programm auszublenden, müssen Sie dann „#define NDEBUG" irgendwo an den Anfang des Quelltexts setzen oder ganz einfach „NDEBUG" in eines der drei „#define"-Symbole schreiben.

Sie können Symbolen aber nicht nur „einfach so" definieren, sondern ihnen auch Werte zuweisen. Ein Symbol-Eintrag wie „N 100" definiert das Symbol „N" als „100". Leider akzeptiert der Compiler bei solchen Vorab-Definitionen immer nur ein Token pro Symbol. Der Eintrag „ANZAHL (17+4)" belegt „ANZAHL" mit dem Zeichen „(„ und ignoriert den Rest einfach. Aber komplexe Definitionen wird man normalerweise ohnehin in den Quelltext schreiben, während derartige „externe" Symboldefinitionen eher für irgendwelche Flags wie „NDEBUG" benutzt werden.

2.5.2.4 Diverse kleine, aber trotzdem wichtige Schalter

Nahezu unübersehbar befinden sich im Compiler-Dialogfenster zwei Spalten mit diversen Switches, also so etwas wie ein virtuelles Mäuseklavier. Die linke Spalte ist mit „Optionen" überschrieben und enthält demzufolge eine mehr oder weniger geordnete Ansammlung von ein- und ausschaltbaren Compileroptionen.

2.5.2.4.1 C++

Mit dem Symbol „C++" können Sie wählen, ob MaxonC++ als echter C++-Compiler oder kastriert im ANSI C-Modus arbeiten soll. Ich hoffe doch stark, daß Sie die C++-Option

nur in Ausnahmefällen ausschalten werden. Sie können diese Umschaltung übrigens auch mit einer „#pragma"-Zeile direkt im Quelltext vornehmen. Näheres dazu erfahren Sie im Kapitel über den Preprozessor.

2.5.2.4.2 Unterbrechen

Mit „Unterbrechen" legen Sie fest, ob der Compiler im erzeugten Code Breakpoints setzen soll. Wenn diese Option angewählt ist, baut er in jede Schleife eine Abfrage der Tastenkombination „Ctrl-C" ein.

2.5.2.4.3 IntMult 32

Der 68000er-Prozessor kann leider keine Langworte multiplizieren und dividieren, sondern nur zwei 16-Bit-Werte zu einem 32-Bit Ergebnis multiplizieren und eine 32-Bit-Zahl durch ein 16-Bit-Wort dividieren, wobei Quotient und Rest jeweils 16 Bit breit sind. Für numerische Anwendungen ist das natürlich nicht so schön, weshalb MaxonC++ optional solche Operationen durch Bibliotheksfunktionen ersetzen kann.

Diese korrekt rechnenden Funktionen werden benutzt, wenn das Symbol „IntMult 32" aktiviert ist, und kosten natürlich einiges an Rechenzeit. Dieser Laufzeit-Aufwand ist aber in der Praxis nicht so gravierend, und für korrekte Ergebnisse ist uns schließlich nichts zu teuer.

2.5.2.4.4 Debug-Datei

Die Compileroption „Debugger-Datei" ist für Sie nur von Interesse, wenn Sie den Maxon Debugger besitzen. Der Compiler legt dann zu jeder Übersetzungseinheit eine Datei an, aus welcher der Debugger umfangreiche Informationen entnehmen kann.

2.5.2.4.5 Symbol-Hunk

Anstelle eines Source-Level-Debuggers wie dem hochkomfortablen Maxon Debugger (MDB) kann man natürlich auch einen symbolischen Debugger wie „MetaScope" benutzen. Um dabei überhaupt etwas wiederzufinden, sollte man sich bei Exe- und Objektdateien sogenannte „Symbol-Hunks" erzeugen lassen. Genau dafür dient die gleichnamige Option.

Wichtig ist, daß Sie „Symbol-Hunk" schon beim Compilieren und nicht erst beim Erzeugen der Exe-Dateien anwählen, um den Compiler zu veranlassen, die notwendigen Informationen zu sammeln und aufzubewahren.

2.5.2.4.6 Asm schreiben

Wenn „Asm schreiben" angewählt ist, erzeugt der Compiler zu jedem Text, den er erfolgreich übersetzt, ein Assembler-Listing. Der Assembler-Quelltext wird in eine Datei geschrieben, deren Name wie der des C-Quelltexts anfängt und mit „.s" endet, also z. B. „hallo.c" zu „hallo.s".

Wenn Sie schon andere Compiler näher kennen, wundern Sie sich vielleicht darüber, daß der Compiler scheinbar einen so optimalen Assembler-Text erzeugt, daß für den Assembler nichts mehr zum Optimieren übrig bleibt. Auch werden Sie sich vielleicht fragen, warum Sie mit dem integrierten Assembler von MaxonC++ nicht selbstgeschriebene Assembler-Programme übersetzen lassen können.

In beiden Fällen handelt es sich um Fehlschlüsse, denn MaxonC++ ist etwas ungewöhnlich implementiert. Der Compiler erzeugt überhaupt keinen Assembler-Quelltext, und es gibt auch keinen Assembler.

Um diese kryptische Bemerkung etwas zu erhellen: Der Compiler erzeugt bei der Codegenerierung einen Zwischencode in einem internen binären Format, das zwar in gewisser Weise wie ein Assembler-Quelltext aufgebaut ist, sonst aber absolut nichts mit einer ASCII-Textdatei zu tun hat. Der „Assembler" ist dann ein ziemlich dröges kleines Prograämmchen, daß diesen Zwischencode in echten Maschinencode übersetzt und dabei noch ein paar kleine Optimierungen durchführt. Diese Low-Level-Optimierungen, zu denen z. B. Short Branches gehören, trägt der Assembler dann auch immer gleich im Zwischencode ein, damit er beim nächsten Pass dasselbe nicht noch einmal optimieren muß.

Und dann gibt es noch einen Programmteil, der noch viel simpler als der „Assembler" ist und den internen Zwischencode in lesbaren Assembler-Quelltext umwandelt. Diese Funktion wird benutzt, wenn die Option „Asm schreiben" angewählt ist, und da die Datei geschrieben wird, nachdem der Pseudo-Assembler seine Arbeit getan hat, sind darin schon alle Optimierungen vorgenommen. Der von MaxonC++ generierte Assembler-Source entspricht also genau dem erzeugten Code. Sie können sich den Reassembler sparen, wenn sie die Codequalität genauer unter die Lupe nehmen wollen.

2.5.2.4.7 Source in Asm

Den oben beschriebenen Quelltext kann man sich etwas übersichtlicher gestalten lassen, indem man zusätzlich die Option „Source in Asm" wählt. Dann wird nämlich der C-Quelltext an jeweils passender Stelle in den Assembler-Source als Kommentar eingefügt, so daß sich die Bedeutung der endlosen Assembler-Listings dem menschlichen Leser einfacher erschließt.

2.5.2.4.8 Optimieren

Guter Code ist mit einem gewissen Aufwand verbunden. Sowohl in Bezug auf den Speicherplatzbedarf des Compilers als auch auf die Übersetzungszeit. Deshalb gibt es die wahlweise ab- oder zuschaltbare Option „Optimieren", die MaxonC++ veranlaßt, die Codeerzeugung etwas schlauer aufzuziehen.

Das Thema „Optimieren" ist nahezu unerschöpflich, und folglich dürfte bei jeder neuen Version von MaxonC++ das eine oder andere Feature hinzukommen. Im Moment macht der Compiler bei aktiver Optimier-Option mindestens folgendes:

- Alle anderweitig entbehrlichen Register werden für lokale Variablen benutzt. Die Entscheidung, welche Variablen in Register geladen werden sollen, fällt über gewichtetes Zählen ihres Auftretens.

- Wenn sich herausstellt, daß eine Funktion keine lokalen Variablen auf dem Stack mehr benötigt (z. B. weil alle in Register umgelegt werden konnten), wird das Einrichten eines Stack-Rahmens mittels „LINK" und „UNLK" unterdrückt.

- Der Assembler führt nicht nur einen, sondern zwei optimierende Passes durch. Damit werden z. B. auch Sprünge nach vorn zu Short-Branches optimiert.

Diverse Mini-Optimierungen, die keinen besonderen Aufwand erfordern, werden natürlich immer durchgeführt und sind nicht abschaltbar. Die Grenze zwischen „vermiedenem schlechtem Code" und einer „Optimierung" ist bekanntlich fließend.

2.5.2.5 Optionale Warnungen und Meldungen

Diverse Sprachkonstrukte sind zwar an sich syntaktisch wie semantisch korrekt, aber trotzdem etwas obskur und möglicherweise Folge eines Versehens des Programmierers. Deshalb gibt es zehn Warnungen, die man je nach Situation und persönlichem Programmierstil einzeln ein- und ausschalten kann. In „MaxonC++" dienen dazu die Switches in der rechten Spalte des Compiler-Dialogfensters.

Im einzelnen gibt es die folgenden Warnungen:

- „Kommentare": Bei verschachtelten Kommentaren, wie „/* Test /* Nahu? */" wird gewarnt. Hier liegt der Verdacht nahe, daß am Ende eines Kommentars einfach das „*/" vergessen wurde.

- „Prototypen": Im C-Modus wird gewarnt, wenn eine Funktion aufgerufen wird, zu der kein Prototyp deklariert wurde. In C++ ist das bekanntlich sowieso ein Fehler.

- „Return' fehlt": Wenn in einer Funktion, deren Ergebnistyp nicht „void" ist, nirgendwo ein „return" auftaucht, ist sicher etwas faul. Deshalb sollte diese Warning immer eingeschaltet sein.

- „Leeranweisung": Eine Anweisung wie „42;" hat keinerlei Wirkung und war folglich wahrscheinlich nicht so gemeint. Ein besonders beliebter Fehler ist auch der Aufruf einer parameterlosen Funktion ohne Argumentliste, etwa „test;" statt „test();".

- „K&R-Parameter": Das „K&R" steht für „Kernighan & Ritchie" und damit für den Vor-ANSI-Standard. Mit dieser Option kann man sich bei Parameterlisten im alten Stil eine Warnung ausgeben lassen.

- „#pragma": Es ist natürlich Sinn und Zweck eines „#pragma", daß der Compiler darin eben keine Fehler meldet und alles ignoriert, was er nicht kennt. Wenn Sie aber nicht bloß fremde Quelltexte übernehmen, sondern mit MaxonC++ selbst entwickeln, ist es sicher sinnvoll, wenn Sie sich auf unbekannte Pragma's hinweisen lassen, schließlich könnte es sich dabei ja auch ganz einfach um einen Tippfehler handeln.

- „Temporarys": Laut C++-2.0-Standard ist es ein Fehler, wenn eine Referenz auf einen nicht-konstanten Typ mit einem Nicht-L-Wert initialisiert wird und dabei ein temporäres Objekt eingeführt werden muß. Da diese Regel im 1.0-Standard noch nicht existierte, gibt es dafür keine „echte" Fehlermeldung, sondern eine abschaltbare Warnung.

Ein Beispiel:

```
void dup(int &ir)
{ ir *= 2; }

void main()
{ int i;
  dup(i);        // OK

  long l;
  dup(l);        // VORSICHT!
}
```

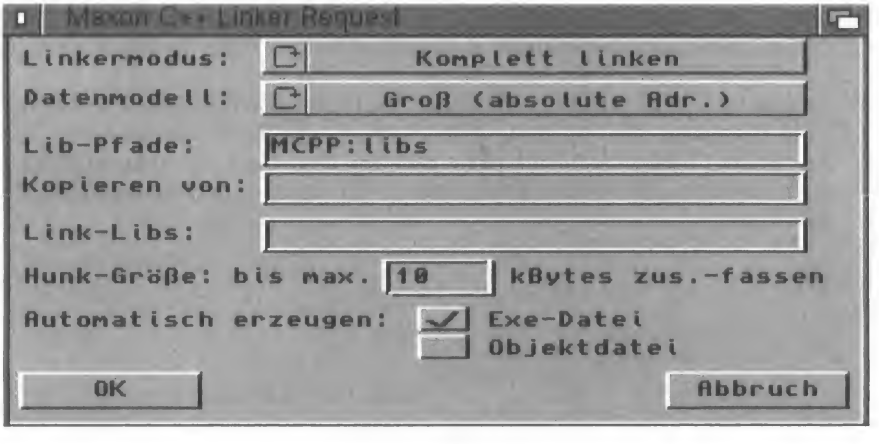
Da hier „l" von „int" nach „long" konvertiert werden muß (in MaxonC++ übrigens eine leere Operation, denn intern sind beides 32Bit-Zahlen), wird ein temporäres Objekt eingeführt, und der Programmierer erlebt höchstwahrscheinlich eine Überraschung, denn eim zweiten „dup"-Aufruf gibt es keinen Seiteneffekt auf das Argument. Die neue C++-Regel ist also ausgesprochen sinnvoll.

- „Unsichere Konv.": Diese Option warnt bei impliziten Typumwandlungen (also ohne Cast) von ganzzahligen in Fließkommatypen.

- „Verdächtiges =": Dabei handelt es sich um das beliebte Konstrukt „if (a=b)..." - korrekt und bei vielen Programmierern beliebt, aber auch ein typischer Anfängerfehler statt „if (a==b)". Wenn die entsprechende Option aktiv ist, wird jeweils eine Warnung ausgegeben, wenn ein „=" auf diese Weise in einem logischen Ausdruck auftritt, also als Bedingung in „if", „while", „for" oder „do" sowie als Operand von „||", „&&" oder „!".

2.5.3 Das Linker-Dialogfenster

Mit dem Menüpunkt „Linker" aus dem „Optionen"-Menü öffnen Sie das Linker-Dialogfenster. Hier finden Sie diverse Optionen zur Steuerung des Linkers.



2.5.3.1 Datenmodelle

Die Umschaltung zwischen „kleinem" und „großem" Datenmodell hat sich eher zufällig in das Linker-Dialogfenster verirrt, da dies eigentlich eher etwas mit der Codeerzeugung und folglich mit dem Compiler zu tun hat.

Zuerst einmal die Frage: Was hat es eigentlich mit diesen Datenmodellen auf sich?

Wenn im vom Compiler generierten Code auf globale, statische Daten zugegriffen werden soll, geschieht dies normalerweise über absolute Adressen. Die Daten selbst können dann irgendwo im Speicher liegen und über beliebig viele Daten-"Hunks" verteilt sein.
Auch die Gesamtgröße dieser Daten ist nur durch die Speichergröße des Rechners begrenzt. Diese kanonische Art der Adressierung bezeichnet man in diesem Zusammenhang als „großes" Datenmodell („Large Data").

Das klappt zwar recht gut und unterliegt auch keinerlei Beschränkungen, aber wie so oft gibt es auch hier Leute, die daran etwas zu mosern haben. Zum Beispiel benötigt eine absolute Adresse im Code immer 32 Bit, also vier Bytes. Das bläht nicht nur die Codegröße auf, sondern kostet auch entsprechend viel Laufzeit. Ferner müssen alle diese Adressen korrigiert werden, wenn das Programm in den Speicher geladen wird, was in der Dateigröße einen gewissen Overhead bedeutet.

Deshalb ist ein anderer Ansatz, das sogenannte „kleine" Datenmodell, in gewisser Hinsicht schlauer. Hier wird ein Prozessorregister, nämlich das Adressregister A4, als Zeiger auf die statischen Daten des Programms benutzt. Die Daten werden jetzt mit einem16 Bit-Offset relativ zu diesem Datenregister adressiert, was jeweils zwei Bytes Code und ein Nichtigstel an Laufzeit spart. Aber jetzt kommen die Nachteile von „Small Data":

- Mit 16 Bit kann man nur 65536 Bytes adressieren - also ist dies die Obergrenze für die globalen Daten des Programms.

- Alle Datenhunks des Programms müssen in einem einzigen Small-Data-Hunk zusammengefaßt werden. Insbesondere unterscheidet man dann nicht mehr zwischen initialisierten Daten („DATA"-Hunk) und den platzsparend abgespeicherten uninitialisierten Daten („BSS-Hunk). Dadurch kann die ausführbare Programmdatei - trotz geringeren Speicherbedarfs des Programms im RAM - überraschend groß werden.

- Bei der Codegenerierung kann das Register A4 nicht für Variablen oder Zwischenergebnisse genutzt werden.

Hinzu kommen noch einige Eigenheiten des Small-Data-Models von MaxonC++:

- Jede Übersetzungseinheit adressiert nur seine „eigenen" Daten relativ zu A4, nicht jedoch irgendwelche importierte Daten. Das hat den Vorteil, daß Code mit „Small Data" und „Large Data" beliebig mischen kann: Von den innerhalb derselben Übersetzungseinheit definierten Daten weiß der Compiler schließlich, ob er sie im „Small Data"-Hunk abgelegt hat, und kann problemlos A4-relativ adressieren. Bei importierten Daten weiß er das nicht und adressiert deshalb im Zweifelsfall absolut. Dieses Konzept schränkt den Nutzen des kleinen Datenmodells bei Programmen, die aus mehreren Modulen bestehen, natürlich etwas ein.

- Das „Small Data"-Konzept von MaxonC++ ist nicht völlig kompatibel zu dem anderer Compiler oder Assembler.

Welche Konsequenz soll man aus diesen Vor- und Nachteilen ziehen? So ganz allgemein wohl gar keine. Im Zweifelsfall sollten Sie einfach ausprobieren, mit welchem Datenmodell Ihr Programm sich besser macht.

Rein praktisch wird das Datenmodell in „MaxonC++" mit dem entsprechenden Symbol im „Linker"-Dialogfenster und beim „CPPC" mit der Option „-gd" bzw. „-gD" eingestellt.

Noch eine Anmerkung zum Schluß: Ich erwähnte oben, daß MaxonC++ bei „large data" das Register A4 munter benutzt. Wie paßt das mit der Aussage zusammen, daß man „small-" und „large-data"-Code beliebig mischen kann? Nun, A4 wird auch bei „großem" Datenmodell nur in solchen Funktionen getrasht, die selbst keine weiteren Funktionen aufrufen und folglich insbesondere keine Funktionen mit A4-relativen Daten. Natürlich wird in solchen Funktionen A4 auch gegebenenfalls auf den Stack gerettet und am Ende der Funktion wieder hergestellt. Damit merkt die gesamte „Umwelt" nichts davon, wenn in einer „large-data-"Funktion A4 mißbraucht wird.

2.5.3.2 Linken oder nicht Linken - das ist hier die Frage

Auch unter „MaxonC++" haben Sie die Möglichkeit, das Linken ganz oder teilweise zu unterlassen, wenn auch nicht mit ganz so vielen Optionen wie beim CPPC. Dazu dient das Symbol „Linkermodus" im Linker-Dialogfenster (wo auch sonst?). Die drei Einstellungen:

- „Komplett linken": Das übersetzte Programm wird ganz normal mit den Bibliotheksroutinen zu einem ausführbaren Programm zusammengebunden oder zumindest wird es versucht.

- „Nicht linken": Es werden keinerlei Bibliotheks- oder sonstige Module hinzugeladen. Der Linker meldet auch keine Fehler, sondern wird überhaupt nicht erst gestartet. Dieser Modus ist recht angenehm, wenn man an einem Executable nicht interessiert ist und lediglich eine Objektdatei haben möchte.

- „Ohne Startup-Code“: Vorsicht, eine Einstellung für Spezialisten - wenn man damit nicht umgehen kann, stürzen die erzeugten Programme gnadenlos ab! Näheres über diese Option dazu finden Sie im Linker-Kapitel, und zwar in Abschnitt 4.6.

2.5.3.3 Bibliotheken

Der Linker von MaxonC++ muß natürlich auch irgendwie wissen, woher er die Standard-Bibliotheken laden muß. Deshalb trägt man die entsprechenden Pfade im „Lib-Pfade“-Symbol des Linker-Dialogfensters ein. Genau wie bei den Includedateien (siehe 2.5.2.1) kann man auch hier bis zu zwei verschiedene Pfade angeben und Dateien dabei automatisch umkopieren lassen.

Von sich aus versucht der Linker immer nur, undefinierte Symbole im „Logfile“ zu suchen und anschließend die entsprechenden Bibliotheksmodule zu laden. Wenn Sie eine Linker-Library, etwa die allseits beliebte „amiga.lib“, einbinden wollen, müssen Sie deren Namen (ohne „.lib“) im Symbol „Link-Libs“ eintragen. Da es nur ein solches Symbol gibt, können Sie darin auch mehrere solcher Bibliotheken durch Leerzeichen getrennt eintragen, z. B. „amiga debug“.

Diese Bibliotheksnamen können wahlweise mit einem vollständigen Pfad oder ganz ohne Pfad angegeben werden. In letzterem Fall sucht der Linker sie wie die „normalen“ Bibliotheksmodule im dafür spezifizierten Verzeichnis.

2.5.3.4 Zusammenfassen von Hunks

Jedes Modul eines Programms besitzt normalerweise einen Code-Hunk und außerdem noch bis zu drei verschiedenen Datenhunks. Bei einem Programm, das aus mehreren Übersetzungseinheiten besteht, kommt da schon so einiges zusammen, einmal ganz zu schweigen von den Bibliotheksmodulen. Damit das mit der Hunkanzahl nicht Überhand nimmt, ist es üblich, gleichartige Hunks zu einem einzigen zusammenzufassen.

Je weniger Hunks ein Programm besitzt, desto kleiner ist die ausführbare Programmdatei, denn jeder Hunk benötigt einen Overhead von ein paar Bytes. Andererseits ist es aber auch nicht schlau, grundsätzlich alle jeweils gleichartigen Hunks zusammenzufassen, denn ein 250 kByte großer Hunk benötigt nun einmal einen ebenso großen, zusammenhängenden Speicherbereich. Bei Rechnern, die knapp an RAM sind, kommt es durchaus vor, daß der Speicher so fragmentiert ist, daß zwar durchaus noch fünf Blöcke zu je 50 kByte, aber kein einziger zusammenhängender 250k-Block frei ist - ein Programm mit einem unangemessen großen Hunk könnte also nicht mehr geladen werden.

Deshalb kann man eine Höchstgrenze wählen, die der Linker beim Kombinieren von Hunks nicht überschreiten soll. Der entsprechende Betrag ist in kByte im entsprechenden Symbol des Linker-Dialogfensters einzutragen. Wenn Sie z. B. aus irgendwelchen Gründen wollen, daß überhaupt keine Hunks zusammengefaßt werden, setzen Sie den Wert einfach auf Null.

In einem Fall nimmt sich der Linker aber auf jeden Fall eine gewisse Freiheit heraus: Bibliotheksmodule werden immer zu Blöcken von 10 kByte kombiniert, auch wenn Sie einen kleineren Wert vorgegeben haben. Das liegt ganz einfach daran, daß die automatisch eingebundenen Bibliotheksfunktionen naturgemäß aus etlichen kleinen Hunks bestehen und es einfach Wahnsinn ist, jeden davon im ausführbaren Programm einzeln auftauchen zu lassen.

2.4.3.5 Schreiben von Dateien

Einer der Vorzüge einer integrierten Umgebung wie bei MaxonC++ besteht darin, daß sich ziemlich alles im RAM und ohne Diskette- bzw. Festplattenzugriffe abspielen kann: Die Quelltexte werden aus dem RAM gelesen und das ausführbare Programm auch dort abgelegt.

Nun gibt es aber bestimmte Gründe dafür, sich stets eine aktuelle ausführbare Programmdatei schreiben zu lassen. Deshalb gibt es im Linker-Dialogfenster ein Symbol das Sie aktivieren können, wenn nach jeder erfolgreichen Übersetzung eine Exe-Datei geschrieben werden soll. Die die Compileroption zum Erzeugen einer Datei für den Maxon Debugger eingeschaltet, so wird diese ebenfalls geschrieben.

Mit einem weiteren Symbol können Sie außerdem Objektdateien automatisch erzeugen lassen. Im Make-Modus geschieht das natürlich sowieso, so daß dieses Symbol dann keine Bedeutung hat.

2.5.4 „MaxonC++"-Systemeinstellungen

Mit dem dritten Dialogfenster aus dem „Optionen"-Menü können diverse Einstellungen des „MaxonC++"-Systems vorgenommen werden.

2.5.4.1 Autosave

Wenn die „Autosave"-Option vom „MaxonC++" eingeschaltet ist, werden beim Start eines Programms aus der integrierten Umgebung automatisch alle Editor-Texte ab-

gespeichert, sofern dies noch nicht geschehen ist - bestimmt eine sinnvolle Maßnahme, zumal man in C stets ohne Netz und doppelten Boden programmiert. Dieser Autosave sollte nicht mit dem zeitabhängigen Autosave des Edward verwechselt werden.

2.5.4.2 Erzeugen von Icons

„Icons" legt fest, ob MaxonC++ beim Schreiben von ausführbaren Dateien, Makefiles und Debugger-Dateien jeweils ein Icon erzeugen soll.

2.5.4.3 „MaxonC++" - ein polyglottes System

MaxonC++ ist zweisprachig konzipiert. Den größte Teil der Benutzeroberfläche von „MaxonC++" kann man zwischen Deutsch und Englisch umschalten, lediglich der Compiler und seine Fehlermeldungen wurden in Englisch gehalten. Mit dem Gagdet „Sprache" kann die gewünschte Sprache ausgewählt werden.

2.5.4.4 Protokollieren von Ein- und Ausgaben

Wenn Sie die Option „Protokoll" einschalten, werden sowohl die Ausgaben des Programms als auch die Eingaben des Benutzers in eine Datei geschrieben. Der Dateiname ist im Texteingabefeld einzutragen, wobei „prt:", also der Drucker, als Defaultwert vorgegeben wird.

2.5.4.5 Aufruf eines externen Assemblers

Das integrierte „Make" kann beliebige externe Compiler und Assembler aufrufen. Wenn eine Quelldatei auf „.asm" endet und die Funktion „Text und Objekt aufnehmen" ausgewählt wird, ist es sogar schlau genug, einen Assembler-Aufruf dafür von selbst zu definieren.

Nun weiß „MaxonC++" natürlich nicht, welchen Assembler Sie verwenden (hoffentlich doch den MaxonASM!), über welchen Pfad er erreicht wird und welche Optionen nötig sind. Deshalb können Sie mit dem Symbol „Externer Assembler" im System-Dialogfenster einen Assembler-Aufruf vorgeben, den das Make defaultmäßig benutzen soll, z. B. „ma [ ] -co" oder „as -c [ ] -w26731 -lj" oder was auch immer. An der Stelle, wo das „[ ]" steht, setzt „MaxonC++" jeweils den Namen der Quelldatei ein.

2.5.4.6 Argumente aus der imaginären Kommandozeile

Beim Start von der Shell kann man Programmen Argumente übergeben, die in C mit den Parametern „argc" und „argv" ausgewertet werden können. Auch beim Programmstart aus der Entwicklungsumgebung von MaxonC++ kann man diese Art der Parameterübergabe simulieren, indem man sie im Texteingabefeld „Argumente" einschaltet. Diese Argumentzeile wird dann an aus „MaxonC++" gestartete Programme übergeben.

2.5.4.7 Compiler- und Laufzeit-Fenster

Bekanntlich öffnet „MaxonC++" für den Compiler sowie für die Programmausführung jeweils ein Fenster, das normalerweise so groß wie der Screen ist. Nun kann man aber seinen Workbench-Screen fast beliebig groß machen, und andererseits wird es auch Leute geben, die gern ein wesentlich kleineres Fenster an einer ganz bestimmten Bildschirmstelle hätten. Deshalb kann man Position und Größe dieser Fenster mit vier kleinen Symbolen im „System"-Dialogfenster festlegen.

Die beiden mit „X" und „Y" beschrifteten Symbole legen naheliegenderweise die Bildschirmposition der linken, oberen Windowecke fest, während Sie in den beiden anderen Symbole die gewünschte Breite und Höhe eintragen können. Als Default wird hier die Bildschirmgröße angenommen.

2.5.4.8 Stackgröße

Wenn ein Programm aus „MaxonC++“ gestartet wird, braucht es natürlich auch einen Stack, und im Zweifelsfall nicht zu wenig davon. Die gewünschte Größe kann ebenfalls im „System“-Dialogfenster eingestellt werden.

Nach Möglichkeit nutzt „MaxonC++“ dafür den freien Platz im Compiler-Workspace aus. Wenn dort nicht mehr genug frei ist, wird eben versucht, den Speicher anzufordern, und wenn auch das fehlschlägt, können Sie das Programm eben nicht starten.

2.5.5 Aktuelles Verzeichnis

Es ist manchmal sinnvoll, wie mit dem „cd“-Befahl das aktuelle Verzeichnis für „MaxonC++“ zu wechseln. Insbesondere ist dieses Verzeichnis wichtig, wenn in einem Quelltext Include-Dateien „so“ in normale Anführungszeichen eingebunden werden, oder wenn ein aus der Umgebung gestartetes Programm auf irgendwelche Dateien zugreift, die es nur bei passendem Verzeichnis findet.

Dafür dient der Menüpunkt „Verzeichnis“ im „Optionen“-Menü. Beachten Sie, daß „MaxonC++“ und der Editor zwei getrennte Prozesse sind und der Verzeichniswechsel sich ausschließlich auf „MaxonC++“ bezieht, und daß „MaxonC++“ im Make-Modus automatisch in das Verzeichnis wechselt, in dem die Projektdatei liegt. In diesem Fall akzeptiert „MaxonC++“ nicht so ohne weiteres den Verzeichniswechsel, sondern fragt Sie mit einem Dialogfenster, ob Sie den Make-Modus verlassen wollen.

2.5.6 Voreinstellungen mit der „Config“-Datei

Natürlich wollen Sie nicht immer, wenn Sie „MaxonC++“ starten, Ihre Lieblingseinstellungen von Hand vornehmen. Deshalb gibt es eine Konfigurationsdatei, die normalerweise im „S:“-Verzeichnis liegt und „MaxonC++.config“ heißt. Es handelt sich hier im eine ASCII-Datei, die Sie mit jedem Editor bearbeiten können.

Jeder Eintrag besteht aus einer Zeile der Form „x = y“. Sie können Kommentare setzen, die ausnahmsweise nicht C++-mäßig mit „//“, sondern mit „;“ eingeleitet werden. Die Groß-/Kleinschreibung spielt hier keine Rolle.

Zunächst einmal gibt es einige Einträge für die diversen Switches, z. B. „Autosave“. Wenn Sie diesem Symbol in der Config-Datei eine „1“ zuweisen, also

```
Autosave = 1
```

ist die Autosave-Option von „MaxonC++“ und damit auch das entsprechende Icon im System-Dialogfenster beim Programmstart automatisch eingeschaltet, während eine Null

```
Autosave = 0
```

die Option ausschaltet. Defaultmäßig, also wenn es keine Config-Datei gibt oder selbige keinen „Autosave“-Eintrag besitzt, ist diese Option übrigens eingeschaltet.

Es gibt noch weitere Switches, die alle genau wie „Autosave“ funktionieren:

„Icon“ entspricht dem gleichnamigen Symbol im System-Dialogfenster und ist per Default beim Start von der Workbench ein- und beim Aufruf aus der Shell ausgeschaltet. Acht weitere Optionen entsprechen den gleichnamigen Switches im Compiler-Dialogfenster:

Option	Symbol-Beschriftung	Defaultwert
Mode	C++	1
Break	Unterbrechen	0
Mult	IntMult 32	1
MDB	Debug-Datei	0
Symb	Symbol-Hunk	0
Asm	Asm schreiben	0
AsmSrc	Source in Asm	0
Opt	Optimieren	1
AutoExe	Exe-Datei (im Linker-Dialog)	0
AutoObj	Objektdatei (im Linker-Dialog)	0

Da wir gerade bei Compiler-Optionen sind: Auch der Compiler-Workspace läßt sich natürlich einstellen, und zwar so:

```
Work = 80
```

80 kByte ist hier auch der Default-Wert.

Der Include-Pfad bzw. die beiden Include-Pfade legt man fest mit

```
Incl = "CPP:Include"
```

bzw.

```
Incl = "ram:CPPinc", "CPP:include"
```

Fehlt noch die Einstellung für die optionalen Warnings. Hier gibt es praktischerweise nicht für jeden Switch eine eigene Option, sondern eine einzelne Zuweisung der Form

```
Warn = "revpt",
```

was übrigens auch die Default-Einstellung darstellt. Jeder Buchstabe steht für eine Warnung, die man eingeschaltet haben möchte. Es gibt folgende leicht einprägsame Kennbuchstaben - übrigens die gleichen Bezeichnungen wie bei der „-w"-Option des CPPC:

Abkürzung für		Symbol-Beschriftung
a	Assignment	Verdächtiges "="
c	Conversion	Unsichere Konv.
e	Empty Statement	Leeranweisung
f	Function prototypes	Prototypen
n	Nested comments	Kommentare
o	Old-Style	K&R-Parameter
p	Pragma	#pragma
r	Return	"return" fehlt
t	Temporary	Temporarys
v	Variable	Var. unbenutzt

Man kann auch zwei Linker-Optionen einstellen, nämlich das Datenmodell und den Lib-Pfad. Ersteres geschieht mit der Option „Data", die auf „L" (für „Large") oder „S" (für „Small") gestezt werden kann, z. B.:

```
Data = S
```

Die Festlegung des Library-Pfads (oder der beiden Library-Pfade) geht syntaktisch genau wie die Include-Pfade:

```
Libs = "CPP:Lib"
```

Ferner wären da noch diverse Systemeinstellungen, etwa die Einstellung der Sprache:

```
Lang = D
```

stellt „Deutsch" ein,

```
Lang = E
```

```
"Englisch".
```

Bei der Protokolldatei gilt es sowohl einen Ein-/Ausschalter als auch einen Dateinamen einzustellen, und zwar mit der Option „Prot“:

```
Prot = 0,"prt:"
```

„prt:“ ist übtigens auch der Default, wenn man hier keinen Dateinamen angibt.

Die Option „CallAsm“ legt den externen Assembler-Aufruf fest, den das „Make“ bei der Funktion „Text und Objekt aufnehmen“ für Assembler-Dateien vorgeben soll, und entspricht dem Eintrag im Symbol „Externer Assembler“ im System-Dialogfenster.

Auch für die simulierten Kommandozeilen-Argumente läßt sich mit der Option „Arg“ ein String vorgeben:

```
Arg = "Testargument!"
```

Da wird es Sie nicht weiter verwundern, wenn ich Ihnen sage, daß auch Position und Größe des Compiler- sowie des Laufzeitfensters festgelegt werden können, und zwar in der kanonischen Reihenfolge X,Y,B,H:

```
Window = 0,10,0,0
```

Obige Einstellung öffnet für Compiler und laufendes Programm jeweils ein Fenster, daß am oberen Rand zehn Punkte freiläßt, sonst aber den gesamten Screen füllt.

Der Assembler-Aufruf, den die Funktion „Text und Objekt aufnehmen“ defaultmäßig verwenden soll, wird mit der Option „CallAsm“ festgelegt, z. B. so:

```
CallAsm = "asm [ ]"
```

Was fehlt noch? Eine Option für den Laufzeitstack. Hier ist sie:

```
Stack = 20
```

Natürlich erfolgt die Angabe in kByte.

Es gibt in der Config-Datei aber auch eine Einstellung, für die es in „MaxonC++“ kein Symbol gibt: Den Namen des Editors. Schließlich muß „MaxonC++“ den Edward nachladen und dafür Pfad und Dateinamen kennen. Die Option heißt naheliegenderweise „Ed“ und ist mit folgendem Default-Wert belegt:

```
Ed = "Edward:Edward"
```

3. MAKE - eine integrierte Projektverwaltung

3.1 Das Prinzip

3.1.1 Wozu eine Projektverwaltung?

Größere Projekte bestehen normalerweise nicht aus einer einzelnen Quelltextdatei, sondern aus einer Vielzahl von Modulen, die vom Linker verbunden werden - zumindestens sollte es so sein. Das nimmt leicht Ausmaße an, die nicht mehr ganz leicht zu überschauen sind. Hinzu kommt, daß es natürlich Unsinn ist, nach jeder Änderung das ganze Projekt zu übersetzen. Auf die Dauer wird es aber lästig, sich immer wieder neu zu überlegen, welches Modul neu übersetzt werden muß, einmal ganz davon abgesehen, daß man eigenartige Überraschungen erleben kann, wenn man dabei etwas vergißt. So entsteht das Bedürfnis, solche Vorgänge zu automatisieren, wozu hat man schließlich einen Computer!

Die populärste Projektverwaltung, die auf so ziemlich allen ernstzunehmenden Rechnern anzutreffen und bei allen Compilersprachen tausendfach bewährt ist, heißt „make". Eigentlich ist „make" selbst eine Art Programmiersprache, und das macht es natürlich nicht leicht, ein komplettes Make in eine graphisch orientierte Entwicklungsumgebung wie die von MaxonC++ zu integrieren. Deshalb unterstützt MaxonC++ nur einen recht begrenzten aber den entscheidenden Teil von „make".

3.1.2 Was ist das überhaupt - ein Projekt?

Fangen wir doch einmal ganz graswurzelmäßig von vorn an: Woraus besteht ein Projekt (immerhin war von einer Projekt-Verwaltung die Rede), und was ist überhaupt ein Projekt?

Also, zu den oben erwähnten Einschränkungen gehört, daß ein Projekt immer genau ein ausführbares Programm darstellt. Wenn man im Rahmen eines Projekts ein Programm in mehreren Versionen erstellen will (man denke da nur an „MaxonC++" und „CPPC", die im wesentlichen ein und dasselbe Programm sind, nur mit unterschiedlichen aufgepfropften Benutzerschnittstellen), muß man eben für jedes einzelne Executable eine eigene Projektdatei erstellen.

OK - ein Projekt ist also ein ausführbares Programm. Und wie kommt man zu einem solchen Programm, sofern es sich nicht gerade um ein Bonsai-Miniatur-Projekt handelt? Richtig, das erwähnte ich schon: Man linkt dafür Objektdateien, also Module, zusammen. Genaugenommen bindet der Linker auch noch fertige Bibliotheksmodule ein, aber

das interessiert uns hier nur höchst periphär. Der Freizeit-Existenzialist stellt sich jedenfalls nun die Frage „Und wo kommen die kleinen Module her?", der Coder weiß es schon: Das hat ausnahmsweise nichts mit Schweinskrams zu tun, sondern die Objektdateien werden von irgendwelchen Compilern erzeugt.

Zu jedem der Module gibt es genau einen ganz besonderen Quelltext, nämlich den Text, aus dem die Objektdatei durch Compilieren entsteht. Um so ein Objekt zu machen, braucht die Projektverwaltung also im wesentlichen eine Information darüber, wie der Quelltext dazu heißt, und außerdem noch ein paar Zusatzinformationen, etwa den Namen des Compilers oder irgendwelche der allseits beliebten Compiler-Optionen.

Aber das ist noch lange nicht alles, denn zu einen normalen Projekt gehören noch einige Texte mehr, normalerweise Includedateien, die von den eigentlichen Quelltexten irgendwo und irgendwie benutzt werden. Das führt uns natürlich zu der Frage, wann und wie eine Objektdatei neu übersetzt wird. Nun, bevor wir alle in Rente gehen, werden wir vielleicht noch das Entwicklungssystem erleben, das Programme on-line compiliert, während man den Quelltext bearbeitet - so, wie es jetzt schon der Kick-Ass tut, etwas Schleichwerbung in eigener Sache muß schließlich erlaubt sein. Vorerst ist es jedenfalls noch so, daß man munter an seinen Quelltexten ‚rumediert und irgendwann entscheidet, daß das Programm für eine erneute Übersetzung reif ist. Dann tritt die bereits mehrfach erwähnte Projektverwaltung „Make" in Aktion und schaut nach, welche Objektdateien für das Projekt nötig sind, und bevor diese zusammengelinkt werden, denkt es scharf darüber nach, welche der Objektdateien neu übersetzt („recompiliert") werden müssen. Da ja der Compiler wenigstens zeitweise mit dem Programmierer Schritt halten soll, ist auch noch halbwegs einsichtig, wann das zu geschehen hat: Richtig genau dann, wenn mindestens einer der Texte, aus denen das Modul entsteht (d. h. Quelltext und Include-Files), seit der letzten Übersetzung verändert wurde. Das läßt sich notfalls leicht an Erstellungsdatum und -uhrzeit der entsprechenden Dateien feststellen.

3.1.3 Was hat das ganze mit „Make" zu tun?

Die Projektverwaltung verwaltet im wesentlichen zweierlei Datenbestände: Texte und Objekte. Bei den Texten handelt es sich im allgemeinen um irgendwelche Quelltexte oder Includedateien, und aus diesen Texten erzeugt der Compiler unter der Kontrolle der Projektverwaltung die Objektdateien. Im letzten Schritt werden diese Objektdateien dann mit den Libraries zu einer ausführbaren Datei gelinkt.

Für jede der Objektdateien braucht das „Make" also Angaben darüber, wie es „gemacht" (daher der Name) werden soll, insbesondere den Namen eines Quelltexts und eines Compilers. Das würde man natürlich auch noch von Hand hinkriegen, und deshalb sagt man der Projektverwaltung auch, unter welchen Umständen ein Objekt compiliert werden soll und wann die alte Version noch einmal verwendet werden kann.

Ein Objekt muß neu übersetzt werden, wenn ein Text, der mit diesem Objekt zu tun hat, verändert wurde. Normalerweise handelt es sich bei diesen Texten um den Quelltext des Moduls sowie die darin benutzten Includedateien. Dazu vergleicht „Make" den Datestamp, also der Zeit der letzten Veränderung einer Datei, der Objektdatei mit denen „ihrer" Texte. Natürlich wird ein Objekt auch dann compiliert, wenn es bisher noch nicht existierte.

3.2 Die Praxis

3.2.1 Erstellen von Projekten

Alle Angaben über die Struktur eines Projekts werden sinnvollerweise in einer Datei festgehalten, dem Makefile. Mit MaxonC++ erstellen Sie diese Datei interaktiv mit dem Pull-Down-Menü „Make". Das Makefile wird dabei ständig aktualisiert, also nach jeder Veränderung neu abgespeichert. Da eine solche Datei normalerweise nicht sehr lang sein wird, dürfte dabei auch beim Diskettenbetrieb keine Verzögerung auftreten.

Am besten probieren Sie das Make-Utility einfach aus, indem Sie sich ein Mini-Projekt aus zwei Modülchen erstellen. Damit fangen Sie an, indem Sie den Menüpunkt „Neues Makefile" anwählen. Nun erscheint das Dateiauswahlfenster und fragt Sie nach dem gewünschten Dateinamen. Meist nennt man die Datei ganz einfach „makefile", jedenfalls dann, wenn man nicht mehr als eine solche Datei in einem Verzeichnis hat. Wenn Sie dagegen an einem vorhandendem Projekt weiterarbeiten wollen, benutzen Sie den Menüpunkt „Makefile laden".

MaxonC++ wechselt immer automatisch in das Verzeichnis, in dem die aktuelle Makedatei liegt. Um Konfusion zu vermeiden, wird die Funktion „Verzeichnis" im Menü „Projekt" gesperrt, wenn ein Makefile geladen oder erzeugt wird.

3.2.2 Hinzufügen von Dateien, umständliche Methode

Zu einem Projekt gehört natürlich mindestens eine Objektdatei, und um eine solche zu erzeugen, braucht man einen Quelltext. Versuchen wir es zunächst auf die umständliche Methode und schreiben einen Text, z. B.

```
#include <stream.h>

void test()
{ cout « "Dies ist ein Test."; }
```

Diesen Text speichern Sie einmal als „test.c“ ab (mit einer Datei ohne Namen kann „Make“ nichts anfangen) und wählen dann den Menüpunkt „Make/Text aus Editor hinzufügen“. Nun wird die Datei dem Projekt hinzugefügt, und „Make“ weiß jetzt, daß „test.c“ irgendwie dazugehört. Sie können auch den Manüpunkt „Text aus Datei hinzufügen“ benutzen, dann müssen Sie die Datei mit dem Dateiauswahlfenster auswählen.

Als nächstes sagen wir dem Make, daß unser Programm aus einem Objekt namens „test.o“ entstehen soll. Also wählen wir den Menüpunkt „Make/Objekt hinzufügen“ und tippen dann im Dateiauswahlfenster den Namen der bisher nicht existierenden Datei „test.o“ ein, vorzugsweise im selben Verzeichnis wie „test.c“ und das Makefile. Und dumm, wie Computer nun einmal sind, müssen wir ihm jetzt auch noch klarmachen, daß „test.o“ entsteht, indem man „test.c“ compiliert.

Dazu dient der Menüpunkt „Compileranweisungen“. Im Dialogfenster, das daraufhin erscheint, wählt man den Namen der Objektdatei aus. Dann erscheint ein weiteres Dialogfenster, in dem man den Compiler-Modus (C++, ANSI C und „unbekannt“) und die Quelltextdatei auswählen kann. Klicken Sie hier einfach „test.c“ als Source an und stellen Sie den Compiler-Schalter auf „C++“.

Wenn Sie jetzt das Programm compilieren (Taste F9 oder Menü „Compiler/Übersetzen“), tritt Make in Aktion, stellt fest, daß „test.o“ compiliert werden muß und bindet anschließend aus diesem Objectfile ein Programm zusammen oder versucht es zumindest, denn es gibt ja noch keine „main“-Funktion. „Make“ funktioniert völlig unabhängig davon, welcher Text sich gerade im Editor befindet.

Wenn „Make“ ein Objekt erzeugt, benutzt es dazu den Compilermodus, der im „Compileranweisungen“-Dialogfenster eingestellt wurde, also ANSI C oder C++. Wenn man hier „None“ einstellt, gibt „Make“ lediglich eine Meldung aus, daß dieses Modul neu zu übersetzen ist.

Dieser Modus wird vor allem dann gebraucht, wenn man in sein Projekt Objektdateien anderer Compiler oder Assembler einbinden will, auch wenn man in diesem Fall meist die Kommandozeilenversion des Compilers vorziehen wird.

Das war jetzt wie gesagt die umständliche Methode. Beim folgenden Miniatur-Modul, das die noch fehlende „main“-Funktion enthält, machen wir es etwas einfacher:

```
#include "test.h"

void main()
{
  test();
}
```

Diesen Text speichern wir einfach als „main.c" ab und wählen den Menüpunkt „Tx.+Obj. hinzufügen". Jetzt macht Make vollautomatisch folgendes:

- Die Datei „main.c" wird in die Liste der Texte aufgenommen.
- Das Objekt „main.o" wird dem Projekt hinzugefügt.
- Eine entsprechende Compileranweisung wird gesetzt, und zwar mit dem Modus, der gerade im Menü „Optionen/Compiler/C++" eingestellt ist.

Falls der Name der Quelldatei jedoch auf „.asm" endet, wird als Modus „Extern" gewählt und ein externer Assembleraufruf entsprechend der Einstellung im System-Dialogfenster vorgegeben. Ferner wird eine Abhängigkeit der Objektdatei von der Quelldatei eingetragen, aber diese Abhängigkeiten werden erst weiter unten erläutert.

Jetzt fehlt dem Projekt nur noch eins: Die Header-Datei „test.h", in der die Funktionen des Moduls „test" deklariert. Die Datei enthält lediglich folgendes:

```
void test();
```

also den Prototypen der Funktion. Wenn Sie jetzt den Compiler starten, müßte das Programm korrekt übersetzt und gelinkt werden.

Das so ziemlich wichtigste Feature der Projektverwaltung haben wir aber bisher nicht benutzt: die Abhängigkeiten. Offensichtlich benutzt „main.o" die Datei „test.h" als Includedatei und ist folglich davon abhängig. Immer, wenn „test.h" verändert wurde, sollte deshalb auch „main.o" sicherheitshalber neu übersetzt werden.

Wählen Sie dazu den Menüpunkt „Abhängigkeiten" und anschließend das fragliche Object-File aus dem daraufhin erscheinenden Dialogfenster aus. Im zweiten Dialogfenster, das sich nun öffnet, klicken Sie das Symbol „test.h" an, und schon weiß „Make" Bescheid.

3.2.3 Weitere Funktionen

Natürlich kommt eines Tages auch der Zeitpunkt, an dem ein Text oder ein Objekt aus einem Projekt ,rausfliegt. Dazu dienen die beiden Menüpunkte „Objekt löschen" und „Text löschen". Letzterer entfernt den Text auch aus sämtlichen Compileranweisungen und Abhängigkeiten, in denen er vorkommt.

Ausgesprochen praktisch ist „Text laden", denn damit kann man sehr bequem einen Text des Projekts in den Editor laden. Dabei wird die Funktion-„Öffnen"des Edward benutzt.

„Objekt markieren" ist in einem gut gepflegten Makefile eigentlich überflüssig, aber wer ist schon perfekt... Wenn Sie in diesem Menü eine Objektdatei anwählen, wird diese beim nächsten Compilieren auf jeden Fall neu übersetzt. Von schlampig definierten Abhängigkeiten einmal abgesehen, braucht man dieses Feature auch, wenn man irgendwelche Compileroptionen verändert (z. B. das ganze Projekt von „Small" auf „Large Data" umgestellt oder die Optimier-Option erstmals eingeschaltet) hat.

Und last not least kann man natürlich auch den Make-Modus wieder verlassen und „normal" weiterarbeiten. Dazu dient natürlich der Menüpunkt „Make verlassen".

3.2.4 Externe Übersetzungen

Ein Projekt kann natürlich auch Module enthalten, die nicht in C oder C++ geschrieben sind oder aus anderen Gründen nicht mit MaxonC++, sondern mit anderen Compilern oder Assemblern übersetzt werden müssen. Außerdem können Sie auch Objektdateien haben, zu denen Sie den Quelltext nicht besitzen, z. B. weil jener Teil des Projekts von einem Kollegen geschrieben wird.

Betrachten wir doch zuerst den Fall, daß Sie eine Objektdatei in Ihrem Projekt haben, die aus einem der genannten Gründe nicht vom MaxonC++ erzeugt werden kann. Den Fall können Sie mit dem in 3.2.2 geschilderten leicht lösen: Sie fügen das Modul mit „Objektdatei aufnehmen" in Ihr Make-File ein, und das war's dann.

Halt, das war's noch lange nicht! MaxonC++ hat auch hier noch ein paar Tricks auf Lager, die das Leben des Programmierers (wieder einmal) etwas angenehmer machen. Sehen Sie sich doch einmal den „Compileranweisungen"-Dialogfenster zu so einer quelltextlosen Datei an: Als Compilermodus ist hier „None" angegeben. Trotzdem können Sie aber einen Quelltext angeben. Ein Quelltext zu einer Objektdatei, die nicht übersetzt wird (jedenfalls nicht vom Maxon Make) ist gleichbedeutend mit einer Abhängigkeit, denn auch diese kann man für so eine Objektdatei setzen. Die Projektverwaltung prüft dann, ob die Objektdatei existiert und ob sie vielleicht älter als ihr Quelltext oder als eine ihrer Abhängigkeits-Dateien ist. Falls ja, wird eine Meldung der Art „Datei xyz.o muß extern recompiliert werden" ausgegeben. Damit wissen Sie, was Sie zu tun haben.

Wenn Sie schon ein wenig mit den Compileranweisungen 'rumgespielt haben, wird Ihnen aufgefallen sein, daß es außer „ANSI C", „C++" und „None" noch einen vierten Modus gibt, nämlich „Extern". Dabei ist dann das bisher gesperrte Texteingabefeld „Anweisung" wählbar, während die Quelltext-Symbolleiste komplett deaktiviert ist. In diesem Modus können Sie eine beliebige CLI-Anweisung (z. B. einen Aufruf eines externen Assemblers) eingeben, nach Möglichkeit natürlich eine solche, die die Objektdatei wirklich erzeugt (auch wenn man hier natürlich auch 'echo „Hello, World!"'

eingeben kann). Das Make öffnet bei der Übersetzung dann ein zusätzliches Textwindow und führt die Anweisung darin aus. Achten Sie aber bitte darauf, daß die externe Anweisung eine Objektdatei und keine ausführbare Programmdatei erzeugt, denn damit kann der Linker nichts anfangen.

Außerdem sollte die Projektverwaltung wissen, wann sie diese externe Übersetzung ausführen lassen soll. Deshalb sollten Sie unbedingt den Quelltext dieses Moduls sowie alle seine Includedateien in das Projekt aufnehmen und die Abhängigkeiten korrekt setzen.

Während Mischungen von C++ mit Pascal, ADA, Occam oder anderen Hochsprachen in der Praxis nicht so häufig vorkommen, muß man doch des öfteren ein Assembler-Modul in sein Programm einfügen. Wenn Sie einen Assembler besitzen, der sich als CLI-Kommando aufrufen läßt, z. B. die entsprechende Version des Maxon Assemblers, liegt es natürlich nahe, den Assembleraufruf auf diese Art und Weise in die Make-Datei aufzunehmen. Und weil das oft genug vorkommt, bietet MaxonC++ Ihnen auch hier ein zusätzliches Stück Komfort: Wenn Sie im Editor einen Text haben, dessen Name auf „.asm" endet - also offensichtlich ein Assembler-Quelltext - und dann die Funktion „Text und Objekt aufnehmen" anwählen, nimmt „MaxonC++" die „.asm"-Quelldatei sowie ihre „.o"-Objektdatei auf und setzt dafür sogar schon den richtigen Assembler-Aufruf. Ferner legt es eine Abhängigkeit der Objektdatei von der Quelldatei fest, ganz wie sich das gehört.

Nun weiß „MaxonC++" natürlich nicht, welchen Assembler Sie verwenden und welche Optionen Sie bevorzugen. Deshalb gibt es im „System"-Dialogfenster ein Texteingabefeld mit der Beschriftung „Externer Assembler", in das man eine Rohform des Assembleraufrufs eintragen kann, z. B. „masm -co [ ]". „MaxonC++" übernimmt bei „Text und Objekt aufnehmen" diesen Aufruf und fügt anstelle des „[ ]" den Quelltextnamen ein. Damit dürfte es kein Problem mehr darstellen, Ihren Lieblingsassembler unter „MaxonC++" aufrufen zu lassen.

3.3 Handgemachte Makedateien

Das Format der Projektdateien von MaxonC++ orientiert sich an den Standard-Makefiles, ist aber natürlich etwas strikter, da in der integrierten Entwicklungsumgebung ja nur ein gewisser Ausschnitt des bekannten Make implementiert werden konnte. Es ist dadurch relativ leicht, ein solches Makefile von Hand zu erstellen oder ein vorhandenes Makefile an MaxonC++ anzupassen. Wenn Sie nicht in der integrierten „MaxonC++"-Umgebung arbeiten, sondern die Option „-m" des CPPC benutzen, bleibt Ihnen sogar nichts anderes übrig, als Make-Dateien grundsätzlich manuell zu erstellen.

Es gibt zwei verschiedene Arten von Zeilen, nämlich Abhängigkeits- und Anweisungszzeilen. Sie unterschieden sich formal dadurch, daß letztere mit einem Leerzeichen beginnen. Auf jede Abhängigkeitszeile kann optional eine Anweisungszeile folgen, aber nicht mehr als eine.

Die erste Abhängigkeitszeile muß immer mit dem Namen der Exe-Datei beginnen, in unserem Beispiel etwa „test". Es ist nicht nötig, weitere Abhängigkeiten in diese Zeile zu schreiben. Normalerweise würde man hier die Namen der Objektdateien angeben, aber MaxonC++ ist eben nicht normal.

Die zugehörige Anweisungszeile muß mit der Kennung „CPPC" beginnen, gefolgt von den Objekten des Projekts. Für unser Beispiel fängt die Makedatei also so an:

```
Test:
  CPPC main.o test.o
```

Nun folgen die Abhängigkeits- und Anweisungszeilen der übrigen Objektdateien:

```
test.o: test.c
  C++ test.c
main.o: main.c test.h
  C++ main.c
```

Die Anweisungszeile beginnt mit „C++", wenn das Modul im C++-Modus übersetzt werden soll, oder mit „cc" für ANSI C-Programme. Bei unbekanntem Modus („None") ist hier das vielsagende „from" zu wählen, sofern man überhaupt das Bedürfnis hat, dafür eine Compileranweisung anzugeben (das ist dann eigentlich nur eine weitere Abhängigkeits-Deklaration). Jede andere Compileranweisung wird als externer Aufruf interpretiert.

Es ist übrigens nicht zwingend erforderlich, dem Make zu sagen, daß ein Objekt von seiner Quelltextdatei abhängig ist, denn das weiß es auch so. Also könnte man das Makefile folgendermaßen abkürzen:

```
Test:
  CPPC main.o test.o
test.o:
  C++ test.c
main.o: test.h
  C++ main.c
```

Aber auch das ist noch nicht optimal: Wenn Sie hinter den Namen des Exe-Files einige „.o"-Dateien angeben, geht Make davon aus, daß diese Objekte, sofern nicht anders

angegeben, aus einer entsprechenden „.c"-Datei entstehen, wobei es den Compilermodus wählt, der gerade als Default im „Optionen"-Menü bzw. über die Kommandozeilenoptionen „-pc"/"-pp" eingestellt ist. Mit diesem Feature läßt sich die Makedatei noch weiter abkürzen:

```
Test: main.o test.o
CPPC main.o test.o
main.o: test.h
```

Ein Projekt besteht aber oft nicht nur aus Objektdateien und Standard-Bibliotheken, sondern benötigen auch Linker-Bibliotheken wie die allseits beliebte „amiga.lib". In „MaxonC++" werden diese bekanntlich im entsprechenden Symbol des Linker-Dialogfensters eingetragen, wobei sie dann gegebenenfalls automatisch in das Makefile aufgenommen werden. Falls Sie die Option „-m" des CPPC benutzen, können Sie solche Angaben wie alle anderen Optionen auch in die Kommandozeile aufnehmen, etwa

```
CPPC -w80 -gO -m makefile -l amiga
```

oder wie auch immer. Sie können die Libraries aber auch direkt in das Makefile eintragen, und zwar als Option hinter der ersten Anweisungszeile (der mit dem „CPPC"):

```
Test: main.o test.o -l amiga
CPPC main.o test.o
main.o: test.h
```

Mit diesen Informationen sollte es für Sie kein Problem mehr darstellen, eine MaxonC++-konforme Projektdatei von Hand zu erstellen.

4. Der Linker

4.1 Prinzipielles

Ein Linker erfüllt im wesentlichen zwei Aufgaben:

- Er fügt Module eines Programms zusammen. Das ist ein nahezu unverzichtbares Feature, denn ab einer gewissen Größe ist es praktisch unmöglich, ein Programm in einem einzigen Modul zu schreiben.

- Anschließend bindet er vordefinierte Bibliotheksroutinen in das Programm ein. Auch das ist eine Aufgabe, die aus dem Programmieralltag nicht wegzudenken ist, denn jeder C- oder C++-Programmierer erwartet ganz selbstverständlich, daß simple Funktionen wie „strcpy" oder aufwendige Operationen wie „printf" in irgendwelchen Bibliotheken vorhanden sind und vom Linker anstandslos eingebunden werden.

Diese beiden Funktionen überschneiden sich allerdings, denn der Unterschied besteht einzig und allein darin, daß man dem Linker ausdrücklich sagt, welche Module er in ein Programm einbinden soll, während er die benutzten Bibliotheksfunktionen weitgehend selbständig hinzulinkt - und, wenn er hinreichend dumm ist, auch noch erheblich mehr - vor allem im Workstation-Bereich gibt es Compiler-/Linker-Systeme, bei denen ein „Hello, World!"-Programm reichlich 50 KByte groß ist, weil der Linker gnadenlos riesige Bibliotheken komplett einbindet.

Das ist es, was man allgemein von einem Linker erwartet, zumal die heute gebräuchlichen Linker immer noch auf Konzepten der 50er Jahre basieren, wie Stroustrup einmal sagte. Auch der Standard-Linker des Amiga, ALink die verbesserte Version, „BLink" oder die diversen Nachbauten dieser Linker tun im großen und ganzen nichts anderes. Auch der integrierte Linker von MaxonC++, der zu Link/BLink kompatibel ist, beherrscht diese beiden Aufgaben, wenn auch zum Teil etwas eigenwillig.
Nun reden wir hier aber von C++ und nicht von solchen Antiquitäten wie Fortran oder C, und die neue Zeit fordert nicht nur neue Compiler, sondern stellt auch neue Anforderungen an die damit verbundenen Linker.

Eine der zahlreichen, auf den ersten Blick eher unscheinbaren Neuerungen, die C++ gegenüber C so auf dem Kasten hat, ist die Initialisierung statischer Daten. In C dürfen globale Variablen ausschließlich mit konstanten Werten initialisiert werden, weshalb eine Deklaration wie

```
int i = 42;
```

weder Compiler noch Linker vor unlösbare Probleme stellt: Die Variable „i" wird im Datenhunk des Programms angelegt, und die entsprechende Speicherstelle bekommt von Anfang an den Wert „42". Schon in dem Augenblick, wenn das Programm in den Speicher geladen wird, steht also an der Stelle, an der sich später einmal die Variable „i" befinden wird, das entsprechende Langwort, und das Schönste daran ist, daß das den Linker überhaupt nicht zu interessieren braucht, denn er bekommt auf diese Weise gar nicht mit, daß es eine solche Initialisierung gibt.

In C++ ist die ganze Angelegenheit schon etwas fortschrittlicher, denn dem Programmierer werden in seinen Initialisierungsdrang keinerlei Zügel mehr angelegt. Das folgende ist zum Beispiel ein korrektes C++-Programm:

```
#include <stdio.h>

int murx = printf("Sachen gibt\'s!!!");

void main() { }
```

Obwohl das Hauptprogramm „main" scheinbar leer ist, macht dieses Programm etwas, denn noch vor der Ausführung von „main" wird die globale Variable „murx" mit dem Ergebnis eines „printf"-Aufrufs initailisiert (das Ergebnis von „printf" ist die Anzahl der ausgegebenen Zeichen oder ein Fehlercode). Also passiert hier schon eine Ausgabe, bevor überhaut das scheinbare Hauptprogramm gestarten wird.

In dieser Form ist das natürlich ein eher krankhafter Fall von Programmverschleierung, aber durch Konstruktoren kommt es in C++ oft zu solchen absolut unkonstanten Initialisierungen. Die ganz besondere Pointe ist, daß jedes Modul solche Initialisierungen besitzen kann. Lange Rede, schwacher Sinn: Der Linker muß in C++ nicht bloß linken, sondern auch dafür sorgen, daß Module korrekt initialisiert werden - ja, nicht nur das: falls globale Variablen einen Destruktor haben, muß er sogar dafür sorgen, daß nach Programmende derartige Destruktoren aufgerufen werden.

Sie werden lachen, aber der Maxon-Linker kann das! Wie das genau vor sich geht, wird an anderer Stelle erläutert. Kommen wir statt dessem noch einmal auf die beiden anfangs erwähnten Aufgaben des Linkers zu sprechen.Wie Sie MaxonC++ dazu bringen, Module zusammenzubinden, dürfte aus den vorhergehenden Kapiteln hoffentlich klargeworden sein (Nämlich in der integrierten Umgebung mit einem Makefile und beim „CPPC" wahlweise mit einem Makefile oder entsprechenden Kommandozeilen-Argumenten). Deshalb widmen die beiden folgenden Abschnitte sich der anderen Funktion, dem Einbinden von Bibliotheksfunktionen:

4.2 Standard-Funktionen und das „Logfile“

Der Linker von MaxonC++ beherrscht das Einbinden von Bibliotheksroutinen erstaunlicherweise gleich auf zwei verschiedene Arten. In diesem Abschnitt soll von der etwas unkonventionelleren Methode die Rede sein, nämlich dem Linken mit Objekt-Dateien über eine Datei namens „Logfile“.

Im großen und ganzen ist eine Bibliothek nichts anderes als eine Sammlung von Modulen mit nützlichen Funktionen. Der Linker geht nun hin und nimmt sich selbsttätig aus dieser Sammlung alle Module, die gerade benötigt werden, und benötigt werden eben jene Objekt-Dateien, in denen Symbole (also z. B. Funktionsnamen) deklariert werden, die im aktuellen Programm benutzt werden, bisher aber nicht definiert wurden.

Es ist deshalb ein naheliegender Gedanke, dem Linker in Form einer Datei mitzuteilen, welche Objektdatei er einlinken soll, wenn ein bestimmtes Symbol sonst nicht bekannt ist. Genau dieses genial einfache, aber zugegebenermassen nicht sehr kompatible Prinzip wurde bei MaxonC++ implementiert: Im „lib“-Verzeichnis, in dem sämtliche Linker-Libraries stehen, gibt es eine Datei namens „logfile“, welche Informationen darüber enthält, was wo steht. Der Dateiname wurde etwas eigenartig gewählt, denn normalerweise versteht man unter einem „log“ so etwas wie ein Protokoll - jeder TeX-User kennt die „.log“-Files. Hier ist das „log“ ganz einfach eine legasthenische Verkürzung von „Katalog“. Wenn ich ehrlich sein soll, habe ich das „logfile“ so genannt, weil das einfach gut klingt und mir gerade nichts besseres einfiel. Wesentlich interessanter dürfte für Sie sein, was in dieser Datei enthalten ist.

Das „logfile“ ist eine ganz gewöhnliche Textdatei, die man leicht mit jedem Editor bearbeiten kann, und besteht aus einer Folge von Einträgen wie

```
Dateiname1
  Symbol1
  Symbol2
  Symbol3
Dateiname2
  Symbol4
usw.
```

Eine Zeichenfolge, die in der ersten Spalte beginnt, stellt also den Namen einer Objektdatei dar, die sich nach Möglichkeit ebenfalls im „libs“-Verzeichnis befinden sollte. Falls jedoch im Dateinamen ein „:“ oder „/“ vorkommen sollte, kann die Objektdatei liegen, wo sie will, Hauptsache der Pfad wurde auch richtig angegeben.

Jedem Dateinamen kann eine nahezu beliebige Folge von Symbolnamen folgen, die sich dadurch auszuzeichnen haben, daß sie in ihrer Zeile um mindestens ein Leerzeichen eingerückt sind. In diesem Zusammenhang sei davor gewarnt, hier „TAB“s zu benutzen, denn die mag der Linker überhaupt nicht.

Zur Steigerung der Anschauung hier ein Auszug aus dem originalen Logfile:

```
startup.o
        __StartUp
dosbase.o
        exit_i
        _exit
        abort_standard
        _DosBase
        _SysBase
        _Filehandle_Std_Input
        _Filehandle_Std_Output
        _cin
        _cout
        _cerr
        _std__in
        _std__out
        _std__err
        _errno
        __Writefile
        __Writefile__NoBuf
wbstartup.o
        _wbmain
stdinput.o
        _getc
        getc__P06stream
        _fgetc
        fgetc__P06stream
        _getchar
        getchar_
        __Readfile
        __Readfile__NoBuf
arg.o
        main_
```

Es gibt hier einige (übrigens in Assembler geschriebene) Objektdateien, in denen bestimmte Linker-Symbole deklariert werden, zumindens behauptet das Logfile das. Wenn also in einem Programm ein Symbol namens „_getc“ referiert wird, weiß der Linker, daß er die Datei „stdinput.o“ aus dem „libs“-Verzeichnis einbinden soll.

Es gibt im wesentlichen für jede Funktion eine eigene Objektdatei. Oft wurden aber aus rein praktischen Erwägungen Funktionen mit gemeinsamen Unterroutinen (z. B. „getc“ und „fgetc“) oder sinngemäß zusammenhängende Funktionen (kein „fopen“ ohne „fclose“) in einer Objektdatei zusammengefaßt. Effektiv wird hier also nur das eingebunden, was auch tatsächlich benutzt wird.

Damit dürfte auch klar sein, wie Sie selbst die Standard-Bibliotheken erweitern können: Kopieren Sie einfach die Objektdatei in das „libs"-Directory, und fügen Sie dem Logfile einen entsprechenden Eintrag hinzu. Wenn Sie sich nicht sicher sind, wie die Linkersymbole zu ihren Routinen heißen - was in C++ ja nicht so ganz trivial ist, da ja die Parameterlisten in den Symbolnamen hineincodiert werden, gibt es zwei Möglichkeiten, diese herauszufinden: Entweder lassen Sie sich ein Assembler-Listing erzeugen und suchen sich dort die „XDEF"-Zeilen heraus, oder Sie versuchen zuerst einmal „einfach so", die selbstgeschriebenen Routinen einzubinden, und merken sich dann, was der Linker nicht finden kann.

4.3 Das Hauptprogramm und seine Argumente

Wenn Sie sich das Logfile durchlesen, erweckt vielleicht der Eintrag „main_" für die Datei „arg.o" Ihr berechtigtes Interesse, auch wenn dieser leider eben nicht besagt, daß man das Hauptprogramm überhaupt nicht mehr selbst schreiben muß. „main_" ist in ++-Linkage der Name des parameterlosen Hauptprogramms. Nun darf man „main „natürlich auch mit den beiden kanonischen Argumenten deklarieren, also so:

```
int main(int, char**)
```

Wird dies im C++-Modus übersetzt, lautet das zugehörige Linkersymbol „main_iPPc", und im C-Modus wird der Programmname für den Linker immer in „_main" übersetzt.

Dieses Dilemma löst MaxonC++ elegant dadurch, daß der Startup-Code zunächst in eine Funktion „main_" einspringt. Wenn der Linker dieses Symbol nicht findet, liegt das entweder daran, daß der Programmierer das Hauptprogramm schlicht vergessen hat, oder es ist eben darauf zurückzuführen, daß „main" mit Parametern oder im C-Modus deklariert wurde. Dem Linker ist dies zunächst einerlei, und da das Symbol „main_" im Startup-Code referenziert, aber nirgendwo definiert wird, sucht er es wie jedes andere Symbol in den Bibliotheken. Dort wird er auch prompt fündig und bindet die Objektdatei „arg.o„ ein. Hinter dem Symbol „main_" verbirgt sich in diesem Bibliotheksmodul eine kleine Routine, die den Argumentvektor „argv" erzeugt und initialisiert, „argc" und den Zeiger auf „argv" auf den Stack ablegt und dann in eine Routine „main_iPPc" einspringt.

Wurde das Hauptprogramm im C++-Modus mit den üblichen Parametern deklariert, trägt es eben jenes Symbol, und die Sache ist für den Linker gegessen. Im C-Modus muß er noch einen weiteren Schritt machen: er bindet zusätzlich das Modul „another_main.o" ein, in dem ein Symbol „main_iPPc" deklariert wird. In dieser Mini-Datei steht nichts anderes als ein Sprung an eine Adresse namens „_main", und auf diese Art und Weise wird auch in C-Programme korrekt eingesprungen.

Das Angenehme an dieser Rumlinkerei ist, daß auf diese Weise der Code für die Berechnung von „argv" in C++ nur dann eingebunden wird, wenn das Programm die Argumente auch auswertet, das spart einige Bytes an Codegröße. Im C-Modus wird dieser Code leider immer eingebunden, und hinzu kommen noch 8 Bytes sowie ein Relokationstabelleneintrag für den zusätzlichen Sprung.

Ach ja, noch was: Es gibt auch eine Bibliotheksfunktion „_wbmain". Wenn Sie also nicht selbst eine Startup-Routine für die Workbench schreiben, nimmt der Linker die aus der Bibliothek. Diese Funktion macht aber nichts anderes als umgehend auszusteigen. MaxonC++ geht davon aus, daß Programme, die nicht eigens dafür geschrieben sind, von der Workbench gestartet zu werden, nicht künstlich Workbench-fähig gemacht werden sollten (etwa durch zwangsweises Öffnen eines Text-Fensters oder ähnlichen Schnickschnack). Daß ein MaxonC++-Programm also nicht ohne weiteres von der workbench startbar ist, ist kein Bug, sondern ein Feature.

4.4 Linker-Bibliotheken

Ich glaube, ich erwähnte schon, daß die Sache mit dem Logfile eine Spezialität von MaxonC++ ist. Der große Vorteil ist dabei, neben der einfachen und flexiblen Erweiterbarkeit, die recht hohe Geschwindigkeit vor allem bei langsameren Speichermedien: Der Linker hat detaillierte Informationen darüber, was er einbinden soll, und muß dann nur noch die benötigten Dateien laden.

Gebräuchlicher sind hingegen Linker-Libraries, bei denen die gesamte Bibliothek in einer Datei zusammengefaßt wird. Der Linker liest sich diese Datei aufmerksam durch und pickt sich die benötigten Teile heraus. Interessant ist dieses Verfahren vor allem deshalb, weil die offiziell von Commodore vertriebene „amiga.lib" in einem solchen standardisierten Format vorliegt - natürlich alles „ALink/BLink"-kompatibel.

Deshalb unterstützt auch der MaxonC++ Linker sinnvollerweise dieses Bibliotheksformat. Man kann eine beliebige Anzahl solcher Bibliotheken angeben, aus denen er dann alles nimmt, was er sonst nicht kriegen kann.

Rein technisch erfolgt die Deklaration der zu untersuchenden Bibliotheken in der Kommandozeilenversion mit der Option „-l" und in der integrierten Umgebung mit einem entsprechenden Eintrag im entsprechenden Symbol des Linker-Dialogfensters. Auch in derMakedatei kann man die gewünschten Bibliotheken in der zweiten Zeile (der Linker-Anweisung mit „CPPC" am Anfang) angeben.

4.5 Initialisierungen und das große Aufräumen

In 4.1 wurde schon mit einem gewissen Stolz darauf hingewiesen, daß der MaxonC++ Linker als extraordinäres Feature auch Module initialisieren kann. Da fragt sich der geneigte Leser natürlich, woher dieses Wunderprogramm weiß, was wie initialisiert werden soll.

Die Lösung dieses Problems ist ganz einfach: Wenn der Linker ein Symbol findet, das mit „_INIT_x_" anfängt - wobei „x" eine Ziffer zwischen 0 und 9 ist - weiß er, daß diese Funktion im Startup-Code aufgerufen werden soll. Umgekehrt werden Funktionen mit dem Anfang „_EXIT_x_" am Programmende aufgerufen. Die Ziffer „x" gibt in beiden Fällen die Priorität der Initialisierung bzw. des Cleanups an: Initialisierungen mit der Nummer 0 werden als erste aufgerufen, die mit 9 als letzte. Bei den „_EXIT_"-Funktionen ist die Reihenfolge umgekehrt. Wenn der Compiler ein Modul erzeugt, das initialisiert werden muß, erzeugt er eine Initialisierungsfunktion mit der Priorität 8, und für eventuelle Destruktoraufrufe wird entsprechend eine Exit-Funktion gleichen Vorrangs generiert.

Der genaue Name der Init- und Exit-Symbole ist belanglos, sie sollten nur im Programm möglichst eindeutig sein. Deshalb kodiert der Compiler den Namen der Objektdatei in die Funktionsnamen, während in den fertigen Libraries (Sie erinnern sich: alle in Assembler geschrieben) möglichst chaotische Namen gewählt wurden, die bestimmt nirgendwo sonst vorkommen, z. B. „_INIT_0_files_o_Yeah_Initialise_me_Babe" im Modul „files.o".

Der Linker erzeugt selbstständig einen kleinen Startup-Code, der erst sämtliche Initialisierungen in der Reihenfolge ihrer Priorität aufruft und dann in den eigentlichen Startup-Code, der für gewöhnlich aus dem Bibliothksmodul „startup.o" eingebunden wird, einspringt. Außerdem generiert der Linker stets eine Funktion, in der sämtliche Exit-Funktionen aufgerufen werden. Die Standard-Funktion „exit" springt dieses Label an.

4.6 Mit und ohne Startup-Code

Manchmal will man den oben beschriebenen Startup-Code nicht haben, zum Beispiel, weil man eine Shared Library programmieren will oder weil einem der Startup-Code einfach nicht paßt. Deshalb gibt es einige Linkeroptionen, die eben diesen unterdrücken.

Zuerst wäre da die schlichte CPPC-Option „-gs" oder die Einstellung „Ohne Startup-Code" im „Linker"-Dialogfenster von „MaxonC++". Bei dieser Option werden die Module des Programms in der angegebenen Reihenfolge gelinkt, die benötigten

Bibliotheksroutinen werden angehängt und als allerletzter Hunk wird eine Sparversion des Startup-Codes erzeugt. Dieser kleine Hunk enthält dann zwei parameterlose Funktionen mit den Labels „_InitModules“ und „_CleanupModules“. Diese sollte man auf jeden Fall am Programmanfang bzw. -ende aufrufen. Von C++ aus muß man sie natürlich in C-Linkage importieren, z. B. so:

```
extern "C"
{ void InitModules();
  void ExitModules();
}
```

Diese Definitionen stehen aber auch im Include-File „<linkerfunc.h>“.

Falls man, das kanonische Beispiel für Linken ohne Startup-Code, eine Shared Library implementiert, sollte **„InitModules()“** die allererste Anweisung in der Open-Routine der Library und **„CleanupModules()“** entsprechend die letzte Anweisung in der Schließroutine sein.

Für fanatische Bytezähler, aber auch für Leute, die CPPC als BLink-Ersatz verwenden möchten, hat die CLI-Version noch zwei schlauere Optionen. Zum einen gibt es da noch „-gsi“. Hier wird der kleine Zusatz-Hunk mit den Funktionen „InitModules“ und „ExitModules“ nur dann erzeugt, wenn eines der beiden Labels auch tatsächlich irgendwo im Programm referiert wird. Dabei sind rätselhafte Programmfehler natürlich schon fast vorprogrammiert, denn man sollte die beiden Funktionen eigentlich vorsichtshalber immer aufrufen, da man ja nie wissen kann, ob irgendwelche Library-Module eine Initialisierung benötigen.

Deshalb sollten Sie „-gsi“ ganz schnell wieder vergessen und stattdessen die Option „-gsw“ verwenden. Diese funktioniert ganz genau wie „-gsi“, gibt aber eine Warnung aus, wenn eine Initialisierung bzw. ein Cleanup nötig ist, aber nicht durchgeführt wird. Im Klartext: Wenn irgendein Modul eine „_INIT_x_“-Funktion besitzt und das Label „_InitModules“ nirgendwo im Programm referiert wird, gibt der Linker so etwas wie „Symbol ‚_InitModules‘ never used“ aus - für den Cleanup analog. Die Option „-gsw“ ist recht praktisch, wenn man sich recht sicher ist, daß nirgendwo im ganzen Programm einschließlich der Standard-Bibliotheken eine dynamische Initialisierung nötig ist, die entsprechenden Funktionsaufrufe deshalb wegläßt und bei Bedarf über seinen Irrtum aufgeklärt werden möchte.

Für echte Freaks ist es natürlich unter ihrer Würde, auf fertige Routinen zurückzugreifen, und so ist ihnen auch der normale Startup-Code suspekt. Es ist nicht weiter schwierig, sich einen völlig eigenen Startup-Code zu schreiben. Linkt man ein Programm ohne Startup-Code und startet es dann, wird in die erste Routine in der ersten Übersetzungseinheit eingesprungen. Es kommt dann also beim Linken wirklich auf die Reihenfolge der Module an. Durch geschickt definierte Registerparameter kann eine solche Funktion

sogar die Shell-Argumente auswerten. Zuvor sollte sie aber die Module initialisieren und im Zweifelsfall mit „GetBaseReg()“ den Zeiger auf den Small-Data-Hunk initialisieren.

Das Beispielprogramm „nostartup.c“ demonstriert dies:

```
// nostartup.c - Demo für Linken ohne Startup-Code//
// Compileraufruf: cppc nostartup.c -gs
// bzw. "Ohne Startup" im M+Plus-Linker-Requester einstellen

#include <linkerfunc.h>
#include <stream.h>

// Die erste Funktion im ersten Hunk wird angesprungen:
void f(register __d0 int len, register __a0 char *par)
// In Registern d0 und a0 werden die Shell-Parameter übergeben
// Diese Register-Belegungen bremsen die Funktion übrigens gut aus,
// deshalb sollte man in dieser Funktion nicht mehr als nötig tun.
{
  // Wichtig, wenn Small Data-Hunks existieren, und sonst auch
  // nicht verkehrt:

  GetBaseReg(); // Initialisierungen, z. B. Dos-Lib öffnen:
  InitModules();
  par[len-1]=0; // Normalerweise kein
                // Nullbyte am Ende
  // Jetzt geht alles wie immer:
  cout << "Es geht auch ohne Startup-Code!" <<
  "\nParameter-Länge: " << len << "\nParameter: " << par
  << "\n";
  // Programmende: Aufräumen, z. B. DosLibrary schlieflen:
  CleanupModules();

  // "exit()" darf ohne Startup-Code nicht benutzt werden!!!
  // Es ist nur so ein sinnvoller Return-Code möglich:
  len = 42;
}
```

5. Die CLI-Version „CPPC“

5.1 Das Konzept

Es soll angeblich tatsächlich Leute geben, denen integrierte Entwicklungsumgebung von MaxonC++ nicht gefällt. Es gibt natürlich auch gute Gründe dafür, z. B. läßt sich „MaxonC++“ schwierig in vorhandene Umgebungen integrieren. Deshalb gibt es eine CLI-Version von MaxonC++, nämlich das Programm „CPPC“.

Eigentlich ist CPPC ebenfalls ein integriertes System, denn es enthält Compiler, Assembler und Linker. Der Unterschied zum „MaxonC++“ liegt also ausschließlich in der Benutzerschnittstelle: CPPC funktioniert wie ein CLI-Befehl und wird über Kommandozeilen-Argumente gesteuert.

Das System wurde ähnlich wie die C-Compiler unter Unix konzipiert, d. h. Compilieren, Assemblieren und Linken erfolgt mit einer einzigen Anweisung (oder optional auch getrennt). Dateinamen, die auf „.o“, „.obj“ oder „.l“ enden, werden als Objectfiles interpretiert und eingelinkt, alles andere betrachtet der Compiler als Namen von Quelltexten.

Ohne besondere Optionen übersetzt der Compiler alle angegebenen Quelltexte (ohne die entsprechenden „.o“-Files zu schreiben), linkt die angegebenen Objectfiles ein und erzeugt eine ausführbare Datei, deren Name sich vom jeweils ersten Dateinamen ableitet:

```
CPPC test.o haupt.c text.c unter.c nochwas.o
```

compiliert die drei Quelltexte „haupt.c“, „text.c“ sowie „unter.c“, erzeugt aber keine Objektdateien, sondern hält das Ergebnis dieser Übersetzungen resident im Speicher. Anschließend lädt er die beiden Objektdateien „test.o“ und „nochwas.o“ dazu und erzeugt daraus eine ausführbare Datei namens „test“.

Bei kleinen Programmen, die aus nur einem Modul bestehen, ist der Compiler also ausgesprochen einfach zu bedienen: Die Anweisung

```
CPPC name.c
```

übersetzt das Programm, linkt es und erzeugt eine ausführbare Datei namens „name“. Einfacher geht es wohl nicht.

Tritt bei der Übersetzung ein Fehler auf, erfolgt eine Ausgabe der Art

```
File "Dateiname.c", Line 17 :
Error: Identifier "ths" not defined.
Type "c" to continue, "a" to abort, "?" to view:
```

Mit „c" können Sie den Fehler übergehen, „?" gibt die fragliche Stelle des Quelltexts aus. Wenn Sie „a" eingeben oder einfach Return drücken, wird der Übersetzungsvorgang abgebrochen. Falls jedoch die Fehlermeldung mit „Warning:" beginnt, ist „continue" der Default, d. h. der Compiler macht weiter, wenn Sie nur Return drücken.

CPPC ist residentfähig, auch wenn einige Shells einen Prüfsummenfehler melden. Dies ist in jedem Fall harmlos, auch wenn CPPC auf eine etwas ungewöhnliche Methode reentrant gemacht wurde: Anhand eines Semaphors sorgt eine geladene Kopie des Programms dafür, daß sie nicht zweimal gleichzeitig läuft.

5.2 Optionen

5.2.1 Allgemeines

Die oben beschriebenen Features allein sind natürlich noch etwas mager. Alle darüber hinaus gehenden Funktionen werden durch Compileroptionen gesteuert, die in der Kommandozeile angegeben werden und jeweils mit einem „-" beginnen.

Um ein wenig Ordnung in dieses umfangreiche Sammelsurium zu bringen, lassen die Optionen sich anhand ihres jeweils ersten Buchstaben in verschiedene Bereiche einteilen:

-a	Assembler-Sourcegenerierung
-b	Debugging
-c	Objektdatei
-d	Symboldefinitionen
-e	Fehlermeldungen
-g	Codegenerierung
-i	Pfade für Includes
-j	Pfade für Libraries
-l	Linker-Libraries
-m	Make
-o	Exe-Datei
-p	Parser
-w	Warnings und Workspacegröße

Bis auf wenige Ausnahmen sind Optionen stets global: Bei einer Zeile der Form

```
CPPC -x text1.c -y text2.c -z
```

werden „text1.c“ und „text2.c“ gleichermaßen mit den Optionen „x“, „y“ und „z“ compiliert. Mit anderen Worten ist die Reihenfolge von Dateinamen und Optionen egal.

5.2.2 Ergebnisse und Ausgaben

5.2.2.1 Assembler-Quelltext

Nicht immer will man aus einem Quelltext direkt eine ausführbare Datei erzeugen. Oft ist es nötig, den Übersetzungsvorgang in einer früheren Phase abzubrechen oder ein zusätzliches Zwischenergebnis zu erzeugen, z. B. ein Assemblerlisting. Dazu dient die Option „-a“.

```
CPPC test.c -a
```

erzeugt beispielsweise nicht nur die ausführbare Datei „test“, sondern auch das Assembler-Listing „test.s“. Die Option „-a“ bricht den Übersetzungsvorgang also nicht vor der Assemblierung ab, sondern erzeugt zusätzlich zu den übrigen Aktionen den Quelltext.

MaxonC++ kann bekanntlich mehrere Texte in einem einzelnen Schritt übersetzen. Die Option „-a“ ist dabei global, d. h. unabhängig davon, vor oder hinter welchem Dateinamen sie steht, wird für alle übersetzten Module eine Assemblerdatei erzeugt.

5.2.2.2 Objektdateien: -c

Die Option „-c“ unterdrückt das Linken (und damit auch die Erzeugung einer ausführbaren Datei) und erzeugt statt dessen eine Objektdatei. Auch diese Option ist, genau wie „-a“ und so ziemlich alle anderen, global:

```
CPPC file1.C -c file2.C
```

erzeugt die beiden Objektdateien „file1.o“ und „file2.o“, und daran ändert sich auch nichts, wenn man das „-c“ an eine andere Position verschiebt.

Die Option „-c“ ist der Schlüssel zur modularen Programmierung: Wenn ein Programm aus einzelnen Übersetzungseinheiten wie „f1.c“, „f2.c“ usw. besteht, übersetzt man mit

```
CPPC -c fN.c
```

immer nur die Module, die verändert wurden, und linkt das Ganze anschließend mit

```
CPPC f1.o f2.o f3.o ...
```

zu einer ausführbaren Datei zusammen.

Eher selten wird man dagegen die Option „-cx" benutzen: Dabei werden zwar auch „.o"-Files erzeugt, aber das Projekt wird trotzdem gelinkt.

5.2.2.3 Ausführbare Programmdatei: -o

Wenn der Linker erfolgreich war und eine ausführbare Datei erzeugen konnte, benennt er diese normalerweise nach dem ersten Dateinamen, der in der Anweisungszeile auftrat, indem er einfach dessen Endung wegläßt. Falls das Ihnen nicht gefällt, können Sie mit der Option „-o" einen anderen Dateinamen angeben, z. B.

```
CPPC quelle.cpp -o Ergebnis
```

5.2.3 Preprozessor

5.2.3.1 Definieren von Symbolen: -d

Mit der Option „-d" kann man „von außen" ein Preprozessor-Symbol definieren. Dabei entspricht ein Argument

```
-d NAME
```

einer Zeile

```
#define NAME
```

am Anfang des Quelltexts. Eine typische Anwendung ist eine Anweisung wie

```
CPPC text.c -d NDEBUG
```

wenn die Includedatei „assert.h" benutzt wird.

Man kann dem Symbol aber auch einen Wert zuweisen, indem man unmittelbar hinter den Bezeichner ein „=" setzt, z. B.

```
-d ANZAHL=4711
```

Als Wert ist hier aber nur ein Token erlaubt, d. h. bei

```
-d N=17+4
```

würde das „+4“ schlicht ignoriert.

5.2.3.2 Includedateien: -i

Der Pfad, über den CPPC die Includedateien sucht, ist mit „C++:include“ vor-eingestellt. Durch einen oder auch mehrere „-i“-Optionen kann dieser Pfad geändert werden: Mit der Option

```
-i CPP:inc
```

wird ausschließlich im Verzeichnis „CPP:inc“ gesucht, und bei

```
-i CPP:include -i CPP:include/amiga
```

durchsucht der Preprozessor nacheinander die beiden Verzeichnisse.

Tritt in einem Dateinamen ein „|“ auf, z. B.

```
-i ram:include|df0:include
```

werden die gelesenen Dateien automatisch vom zweiten in den ersten Pfad umkopiert, wie es am Ende von Abschnitt 2.5.2.1 beschrieben wurde.

5.2.4 Compiler-Modus: -pc und -pp

Bekanntlich läßt sich der MaxonC++ Compiler in den ANSI C-Modus umschalten. Beim CPPC dient dazu die Option „-pc“, während „-pp“ wieder in den als Default eingeschalteten C++-Modus zurückschaltet.

Im Gegensatz zu allen übrigen Optionen ist es hier erforderlich, daß der Compilermodus vor dem jeweiligen Quelltextnamen steht, z. B. übersetzt

```
CPPC -pc a.c -pp b.c
```

die Datei „a.c“ im ANSI-Modus und „b.c“ als richtiger C++-Compiler.

5.2.5 Codeerzeugung

5.2.5.1 Breakpoints setzen: -gb

Wenn die Option „-gb“ benutzt wird, setzt der Compiler in alle Schleifen und sonstige schleifenverdächtige Konstrukte eine Abfrage des Break-Bits, das z. B. bei Betätigung der Tastenkombination Ctrl-C gesetzt wird. Dadurch ist es möglich, so erzeugte Programme jederzeit bequem abzubrechen, allerdings kostet diese Option auch einiges an Laufzeit.

5.2.5.2 Datenmodell: -gd und -gD

Die Option „-gd“ schaltet auf „Small Data“, „-gD“ auf „Large Data“ (siehe 2.5.3.1). „Large“ ist Default.

Wenn das aber schon defaultmäßig so ist - wofür existiert dann die Option „-gD“? Nun, es kann ja sein, daß in den „CPPOPT“-Voreinstellungen bereits „-gd“ voreingestellt wurde. Dann brauchen Sie „-gD“, um trotzdem auf „Large Data“ zu schalten, ohne die Environment-Variable zu ändern.

5.2.5.3 Zusammenfassen von Hunks: -gh

Mit der Option „-gh“ legen Sie die Höchstgrenze in kByte fest, bis zu der der Linker gleichartige Hunks verbindet (siehe 2.5.3.4), zum Beispiel: „-gh20“ für 20 kByte. Default ist 50.

5.2.5.4 Ganzzahlige Arithmetik: -gm

Wie bereits unter 2.5.2.4.3 beschrieben, hat der 68000-Prozessor gewisse Schwierigkeiten mit der Multiplikation und Division von 32-Bit-Zahlen. Deshalb benutzt MaxonC++ dafür normalerweise korrekt rechnende Bibliotheksfunktionen, die aber recht langsam sind.

Wenn Sie das Flag „-gm“ setzen, benutzt CPPC trotzdem die normalem Rechenoperationen des Prozessors.

5.2.5.5 Optimierungen: -go und -gO

Die zusätzlichen Optimierungen (2.5.2.4.8) werden mit der Option „-go" aus- und mit „-gO" eingeschaltet. Defaultmäßig sind sie aus.

5.2.5.6 Linken ohne Startup-Code: -gs, -gsi und -gsw

Wie in Abschnitt 4.6 beschrieben, läßt „-gs" beim Linken den Startup-Code weg, generiert dafür aber zwei explizit aufrufbare Funktionen „_InitModules" und „_CleanupModules". Bei „-gsi" werden diese nur erzeugt, wenn sie irgendwo im Projekt referiert werden, und „-gsw" gibt zusätzlich eine Warnung aus, wenn eine der beiden Funktionen nicht leer ist, aber trotzdem nirgendwo aufgerufen wird.

5.2.6 Debugger

5.2.6.1 Maxon Debugger: -b und -bi

Wenn Sie den Maxon Debugger besitzen, können Sie mit der Option „-b" die notwendigen Debugger-Dateien („.mdbi" und „.mdbg") erzeugen lassen. Die Option „-bi" schreibt zusätzlich ein Icon für die „.mdbg"-Datei.

5.2.6.2 Symbolischer Debugger: -bs

Die Option „-bs" erzeugt Symboltabellen-Hunks für symbolische Debugger (siehe 2.5.2.4.5).

5.2.7 Warnungen und Fehler

5.2.7.1 Alternative Fehlerbehandlung: -e und -en

Um die Fehlerbehandlung zu mechanisieren, z. B. auch in einem fremden Editor die Fehlerstellen automatisch anspringen zu können, bietet MaxonC++ die Option „-e". Dabei werden Fehlermeldungen in der Form

```
<Dateiname>, Line <Zeilen> / <Spalte>:
<Fehlermeldung>
```

ausgegeben, und die Abfrage „Type ‚c' to continue..." entfällt ersatzlos. Der Compiler bricht spätestens nach der zehnten Warnung oder Fehlermeldung ab. Diesen Wert können Sie verändern, indem Sie die Anzahl direkt an die Option „-e" anhängen, z. B.

```
CPPC totalfalsch.c -e20
```

Nach 50 Fehlermeldungen weigert MaxonC++ sich aber auf jeden Fall weiterzumachen.

Die Option „-en" ist identisch mit „-e", unterdrückt aber zusätzlich noch diverse Statusausgaben.

5.2.7.2 Optionale Warnungen: -w

Die unter 2.5.2.5 beschriebenen, zusätzlichen Warnungen gibt es natürlich auch beim CPPC. Man schaltet eine Warnung ein, indem man ihren Kennbuchstaben GROSS hinter das „-w" setzt, während ein kleiner Buchstabe eine Warnung ausschaltet.

Dies sind die Warnungen und ihre Kennbuchstaben:

a (Assignment): Operator „=" taucht in Bedingung auf - vielleicht der bekannte „=="-Anfänger-Fehler?

c (Conversion): Unsichere Typkonvertierung ohne Cast, z. B. double -> int

e (Empty Statement): Anweisung hat keinen Code, z. B. „42;"

f (Function prototypes): Funktion ohne Prototyp aufgerufen (nur in C sinnvoll, da in C++ sowieso ein Error)

n (Nested comments): „/*" in Kommentar

o (Old-Style): Über Parameterlisten im alten ‚K&R'-Stil meckern (nur in C sinnvoll, da in C++ sowieso ein Error)

p (Pragma): Unbekanntes „#pragma"

r (Return): Funktion mit Rückgabewert hat kein „return"

t (Temporary): Temporäres Objekt wurde eingeführt, um ein nicht-"const"-Referenz zu initialisieren - was seit dem 2.0-Standard eigentlich sogar ein Error ist.

v (Variable): Lokale Variable wird nie benutzt bzw. benutzt, aber nie initialisiert.

Defaultmäßig sind „-wEPRT“ aktiviert. Um z. B. Warnungen bei fehlenden Prototypen ein- und solche bei unbekannten „#pragma“ auszuschalten, setzen Sie einfach die Option „-wFp“ oder alternativ die beiden Optionen „-wF -wp“.

5.2.8 Linker

5.2.8.1 Library-Pfade: -j

Der Pfad, über den der Linker die Bibliotheken lädt, ist mit „C++:lib“ voreingestellt und kann mit der Option „-j“ verändert werden, z. B. „-j dh1:cpp/lib“. Genau wie bei der Option „-i“ (5.2.3.2) kann auch hier ein mit „|“ getrennter Doppelpfad wie z. B „ram:clib|C++:lib“ angegeben werden, um die benutzten Dateien automatisch auf ein schnelleres Speichermedium umkopieren zu können. Im Gegensatz zu den Include-Pfaden kann für Libraries aber immer nur ein Pfad bzw. Doppelpfad definiert werden.

5.2.8.2 Linker-Libraries: -l

Um Linker-Libraries (2.5.3.3) einbinden zu lassen, setzt man ihren Namen einfach hinter die Option „-l“, z. B. wird bei

```
CPPC ami.c -lamiga
```

die „amiga.lib“ eingebunden. Hinter dem „-l“ darf optional auch ein Leerzeichen stehen, und natürlich kann man auch mehrere Libraries einbinden.

5.2.9 Die integrierte Projektverwaltung: -m

Das Make-Utility gibt es nicht nur im der integrierten Umgebung „MaxonC++“, sondern auch im CPPC. Man übersetzt anhand eines Makefiles, indem mn die Option „-m“ benutzt, z. B.

```
CPPC -m makefile
```

Die Make-Datei muß die in Abschnitt 3.3 beschriebene Form besitzen. Man kann zusätzliche globale Optionen für die Übersetzung angeben, z. B.

```
CPPC -m makefile -w80 -gO -gd
```

etc. Namen zusätzlicher Text- und/oder Objektdateien werden im Make-Modus in der Kommandozeile naheliegenderweise nicht akzeptiert.

5.2.10 Arbeitsspeicher: -w

MaxonC++ alloziert zwar den Speicherplatz für die meisten internen Angelegeheiten automatisch und dynamisch, aber bei der Codegenerierung wird trotzdem ein zusammenhängender Speicherbereich benötigt, dessen Größe mit 50 kByte voreingestellt ist. Sollte bei einem umfangreichen Programm einmal eine Meldung wie

```
Workspace overflow!
```

ausgegeben werden, sollten Sie die Größe dieses Speicherbereichs mit der Option „-w" behutsam erhöhen, z. B. „-w80" für 80 kByte.

5.3 Voreinstellungen mit „CPPOPT"

Wenn es Ihnen zu mühsam ist, bei jedem Compileraufruf Ihre Lieblings-Optionen von Hand einzutippen, können Sie sich natürlich in einer geeigneten Shell ein passendes Alias definieren, z. B. haben Sie sich durch

```
alias cc2 "CPPC -pc -w100 "
```

einen C-Compiler mit großem Workspace geschaffen. Eleganter geht dies aber mit der Environment-Variablen „CPPOPT", die aber erst ab Betriebssystemversion 2.0 unterstützt wird:

```
setenv CPPOPT -pc -w100
```

stellt die beiden genanntem Optionen als Default ein. Die Variable wird so behandelt, als würde ihr Wert an den Anfang der Kommandozeilenargumente gestellt. Deshalb können Sie hier theoretisch nicht nur irgendwelche Optionen wählen, sondern z. B. auch mit

```
setenv CPPOPT test.c
```

zusätzliche Quelltexte oder Objektdateien angeben. Dies dürfte aber nicht unbedingt sinnvoll sein.

5.4 Zusammenfassung

Es folgt noch einmal eine kurze, alphabetisch geordnete Übersicht über alle CPPC-Optionen:

-a erzeugt für jedes übersetzte Textfile einen Assemblerquelltext.

-as C-Quelltext wird in Assembler-Source als Kommentar eingefügt.

-b Debug-Info-Datei erzeugen.

-bi Wie „-b", aber Icon für „.mdbg"-File erzeugen.

-bs Hunk „Symbol" für symbolischen Debugger erzeugen.

-c Die Quelltexte werden compiliert und die entsprechenden „.o"-Dateien erzeugt. Das Programm wird nicht gelinkt.

-cx Wie „-c", aber das ganze Projekt wird trotzdem gelinkt und ein Executable (daher das „x") erzeugt.

-d NAME
definiert ein Preprozessor-Symbol. Optional kann unmittelbar hinter den Namen ein „=" und der gewünschte Wert (nur ein Token) gesetzt werden, andernfalls ist das Symbol als „leer" definiert.

-e[N] Fehlermeldungen werden ausgegeben, ohne daß Interaktion des Benutzers verlangt wird. Spätestens nach dem „N"-ten Fehler wird abgebrochen (Default ist 10).

-gb Breakpoints setzen (defaultmäßig OFF).

-gd Datenmodell: Small Data.

-gD schaltet auf „Large Data", was auch Default ist.

-gh xxx
legt die maximale Größe (in kByte) fest, bis zu der gleichartige Hunks vom Linker zusammengefaßt werden. Default ist 50.

-gm MULS/DIVS-Operationen des Prozessors benutzen (Default: Bibliotheksfunktionen für korrekte 32-Bit-Arithmetik).

-go schaltet Optimierungen aus.

-gO schaltet Optimierungen ein.

Optimierungen bisher (neben Details):
- Einführen von Register-Variablen
- Ein optimierender Assembler-Pass mehr
- „link/unlk" werden nach Möglichkeit ausgelassen

-gs Ohne Startup-Code linken

-gsi Wie „-gs", aber Hunk mit „InitModules" und „CleanupModules" nur erzeugen, wenn diese Symbole auch irgendwo benutzt werden (praktisch, wenn man CPPC als schlichten Linker z. B. für Assembler-Projekte mißbraucht).

-gsw Wie „-gsi", aber zusätzlich Warnung ausgeben, wenn „InitModules" oder „CleanupModules" zwar nicht leer sind, aber trotzdem nirgendwo benutzt werden. Dann hat offenbar der Programmierer einer Shared Library o. ä. diese Aufrufe vergessen.

-i xxx

Der Pfad, über den CPPC die Includefiles sucht, ist mit „C++:include" voreingestellt. Durch einen oder auch mehrere „-i"-Argumente kann dieser Pfad geändert werden. Tritt in einem Dateinamen ein „|" auf, werden die gelesenen Files automatisch vom zweiten in den ersten Pfad umkopiert, z. B. „-i ram:includeldf0:include"

-j xxx

Analog zu „-i" stellt diese Option den Pfad bzw. den Doppelpfad, über den die Libraries gesucht werden, ein. Es kann hier aber nur ein Pfad bzw. ein Doppelpfad (mit „|") angegeben werden. Default ist „C++:lib".

-l xxx

Die Library „xxx.lib" wird aus dem aktuellen Libs-Verzeichnis eingebunden.

-m xxx übersetzt nach Makefile „xxx".

-o xxx legt einen anderen Namen für das Exe-File fest.

-pc schaltet auf Default-Mode „ANSI C".

-pp C++ als Default-Mode. „-pc" und „-pp" müssen stets VOR dem jeweiligen Quelltext stehen. „CPPC -pc a.c -pp b.c" übersetzt „a.c" als ANSI und „b.c" als C++.

-w Warnings ausgeben. Es kann eine beliebige Folge von Groß- und Kleinbuchstaben folgen, wobei ein GROSSER Buchstabe die Warnung einschaltet.

Folgende Warnungen gibt es:

a (Assignment): Operator „=" taucht in Bedingung auf - vielleicht der bekannte „=="-Anfänger-Fehler?

c (Conversion): Unsichere Typkonvertierung ohne Cast, z. B. double -> int

e (Empty Statement): Anweisung hat keinen Code, z. B. „42;"

f (Function prototypes): Funktion ohne Prototyp aufgerufen (nur in C sinnvoll, da in C++ sowieso ein Error)

n (Nested comments): „/*" in Kommentar

o (Old-Style): Über Parameterlisten im alten ‚K&R'-Stil meckern (nur in C sinnvoll, da in C++ sowieso ein Error)

p (Pragma): Unbekanntes „#pragma"

r (Return): Funktion mit Rückgabewert hat kein „return".

t (Temporary): Temporäres Objek wurde eingeführt, um ein nicht-"const"-Referenz zu initialisieren - was seit dem 2.0-Standard eigentlich sogar ein Error ist.

v (Variable): lLokale Variable wird nie benutzt bzw. benutzt, aber nie initialisiert.

Defaultmäßig sind „-wEPRT" aktiviert.

-wN Setzt den compilerinternen Workspace auf „N" KByte. Default sind 50 KByte, und das sollte reichen.

MAXONEDWARD

Nicht einfach nur ein Editor!

MAXON
computer

Inhaltsverzeichnis

Prolog 7

Einleitung - Setzen Sie Ihre Gefahr-O-Sensomatic-Sonnenbrille auf 8

I. Installation von Edward auf der Festplatte 9

II. Aufruf von Edward 9
1) Start von der Workbench 9
2) Start aus dem CLI 10

III. Eingabe und Bearbeitung von Texten 11
1) Die Status-Anzeige 11
2) BACKSPACE und DELETE 12
3) UndoLine-Funktion 12
4) RETURN, ENTER und TAB 12
5) Die Cursor-Tasten und der Ziffernblock 13
6) Der Schieberegler 14
7) Die Maus 14

IV. Text 15
1) Erzeugen von Texten 15
2) Wahl eines bestimmten Textes 16
3) Freigeben eines Textes 16
4) Löschen des aktuellen Textes 17
5) Drucken eines Textes 17

V. Dateioperationen 18
1) Speichern von Texten 18
2) Laden von Texten 19

VI. Blockoperationen 20
1) Cut, Copy, Paste 20
2) Spaltenblöcke 21
3) Laden, Speichern und Drucken 23
4) Einrücken 23
5) Umwandeln in Groß- bzw. Kleinbuchstaben 24

VII. Suchen und Ersetzen 25
1) Das „Find"-Formular 25
2) Das „Find & Replace"-Formular 26
3) Struktur-Check (Klammer-Check) 27
4) Compiler Error-Datei interpretieren, Fehler anspringen 28
5) Auf bestimmte Zeilen positionieren 29

VIII. Strukturieren von Texten, Lesezeichen 30
1) Erzeuge Falte, Zeige Falte, Entferne Falte, Zeige alle Falten, Schließe alle Falten 30
2) Auszeichnen von Zeilen 32
3) Cursor-Bewegung 33
4) Lesezeichen (Bookmarks) 33

IX. Das Edit-Prefs Formular 34

X. „Word-Wrap" 36

XI. Makros 37
1) Makros aufzeichnen & abspielen 37
2) Makros speichern & abspielen 38

XII. Fenster 40
1) Erzeugen von Fenstern 40
2) Wahl eines bestimmten Fensters 41
3) Schließen von Fenstern 42
4) Arbeiten mit mehreren Ansichten eines Textes 42

XIII. Verlassen des Editors 44
1) Unterbrechen der Arbeit 44
2) Verlassen des Editors 45

XIV. Iconify & HotKey 46
1) Iconify 46

2) HotKey 46

XV. Der Kommandozeilen-Interpreter 47

XVI. Globale Einstellungen 48
1) Das „Globale Einstellungen"-Formular 48
2) Das „Seitenformat"-Formular 51

XX. Benutzerdefinierte Menüs, Tastaturbelegung und HotKey 53
1) Kurze Einführung in die EBNF-Darstellung 53
2) Definition eigener Menüs 55
3) Definition der Tastaturbelegung 56
4) Definition des HotKeys 59
5) Einladen einer neuen Definition 60

XXI. ARexx - Kommunikation mit Edward 61

XXII. Zurücknehmen von Text-Änderungen 62

Anhang A) „Die Kommandos des Editors" 64
I. Eingabe und Bearbeitung von Texten 65
II. Verwalten von Texten 68
III. Dateioperationen 70
IV. Blockoperationen 71
V. Suchen und Ersetzen 74
VI. Strukturieren von Texten 76
VII. Ändern von Optionen, die den aktuellen Text betreffen 78

VIII. „Word-Wrap“ 79
XI. Makros 80
X. Fenster 81
XI. Verlassen des Editors 82
XII. Formulare 83
XIII. Der Kommandozeilen-Interpreter 84
XVI. Ändern von globalen Einstellungen 85
XV. ARexx - Kommunikation mit Edward 86
2) Laden von Texten aus Edward heraus 89
XVI. UnDo/ReDo - Zurücknehmen von Text-Änderungen 90
XVII. Verschiedenes 91

Anhang B) Fehlermeldungen. 92
1) Informationen 92
2) Allgemeine Fehlermeldungen 93
3) Fehlermeldungen mit Abfragen 97
4) Meldungen mit Abfragen 97
5) Fehler bei Betriebssystemoperationen 100
6) Eingabefehler 102
6) Fehler in den Definitionsdateien 103
7) Fehler beim Analysieren von Kommandos 105
8) Fehlermeldungen beim Auswerten von Kommandos 106
Sekundärliteratur 107

Index 108

Prolog

Edward - Der königliche Editor / Ein Roman von Thomas E. Wieger

Und es begab sich einmal vor langer, langer Zeit, in einem kleinen unbeugsamen Dorf, irgendwo in Deutschland, wo ein begnadeter (hüstel) Coder hauste und - seitdem die Stromversorgung in besagtem Dorfe einigermaßen stabil geworden war - in seinem Developers-Cave, tief drunten im Keller des finsteren Hauses, in welchem er wohnte, vor sich hinprogrammierte, daß jener Coder, dessen Namen wir hier noch nicht nennen möchten und aus rechtlichen Gründen auch nicht nennen dürfen, auf einen Gedanken kam (hört, hört).
Ein Gedanke, hoppla, wie konnte der nur Platz finden in dem mit seltsamen Bytes angefüllten Hirn des Coders. Nun, darüber wollen wir, liebe Leser, uns nicht den Kopf zerbrechen, warten wir lieber ab, was der Gedanke zu sagen hatte.
AE (der auktoriale Erzähler): „Nun sehr geehrter Gedanke, dann nehmen Sie doch bitte mit wenigen Worten Stellung zu Ihrem Inhalt."
GG (der göttliche Gedanke): „Ähm, tja, also ich habe vor, das gesamte Hirn des Coders mit einer einzigen Absicht zu überschwemmen. Er wird nur mich im Kopf haben, für andere Gedanken wird gar kein Platz mehr sein. Er kann nicht mehr schlafen, vom Essen wollen wir gar nicht erst reden. Nur noch ich werde ihn in seinem ganzen Handeln lenken. Tja, ja, lieber AE, das hätten Sie wohl nicht gedacht!"
AE : „Durchaus, sehr bemerkenswert. Aber wollen wir nicht nun unser Augenmerk auf Ihren besonderen Inhalt lenken. Bitte.."
GG : „Ja. Wie wunderbar. Mein Inhalt. Oh, dieses göttliche schulterlange braune Haar. Diese wunderbaren grünen Augen, in denen man wie in unergründlichen, tiefdunklen Gebirgsseen versinken könnte. Das sanfte und zarte Lächeln, das manchmal über ihr Antlitz huscht.. Ohh,... seufz... Huch, momentmal, lieber AE, Sie drücken mir ja die Luft ab, und was soll denn dieses Messer, Sie werden sich noch verletzen. Hilfeeee..."
AE (wischt sich das Blut von den Händen) : „Werter Leser, verzeihen Sie dieses unangenehme Intermezzo, selbstverständlich dachte der geniale Keller-Coder nicht an irgendwelche Frauen, vielleicht weiß er noch nicht einmal, was Frauen sind. Dieser Gedanke wollte sich nur auf äußerst lästige Weise in meine Geschichte drängen. Reine Prahlsucht war es, die ihn solchermaßen um unsere Gunst buhlen ließ."
Nun denn. Wenden wir unser Augenmerk nun dem wahren Gedanken zu, der fürwar nicht sonderlich originell war. Der Coder hatte sich, wie tausende Coder vor ihm, und tausende, die ihm noch folgen werden, dazu entschieden, einen Editor zu programmieren. Nun ja, sicher, keine besondere Idee. Aber da der Coder von dieser Idee vollständig besessen war (er hatte ja keine weitere Idee, und der Gedanke an die unbekannte braunhaarige Frau war auf grausamste Weise getötet worden), erstand im Laufe der folgenden Tage, Wochen und Monate ein Editor, der sich durch außerordentliche Merkmale auszeichnete.

Und so sprach eines Tages eine Stimme zu dem Coder mit tiefem Donnerhall : „Coder, ich rufe dich."
Coder : „Huch, hoppla, wasn das fürn Sample?"
Die Stimme : „Idiot, ich bin der obermächtige Guru, und du wirst mir Gehör schenken!"
Coder : „Au man..."
Die Stimme : „Ich habe dein Werk betrachtet und gesehen, daß es gut ist. Also hau' dich mal aufs Ohr. Ach noch was. Bevor dein Kind auszieht, um die Welt zu erobern, gib ihm einen Namen. Und da es der Sohn eines Königs ist, so braucht es einen königlichen Namen. Nenn es : EDWARD."
Die göttlichen Fanfaren erschallen.

Einleitung - Setzen Sie Ihre Gefahr-O-Sensomatic-Sonnenbrille auf

Hallo, ich bin der Autor dieser Anleitung und ich werde Sie mit Edward vertraut machen. Sollten Sie nur wenig Ahnung haben, so lesen Sie bitte die folgenden Kapitel sorgfältig durch. Mit kleinen Übungen und vielen Erläuterungen werden Sie alle Funktionen Edwards kennenlernen.
Am wichtigsten „don't panic", also keine Aufregung, gemeinsam werden wir Edward schon in den Griff bekommen und dafür sorgen, daß wir der „Chef" sind und nicht der Computer. Im Notfall legen Sie sich einfach ein Handtuch auf den Kopf.
Falls Sie schon etwas mehr Ahnung vor sich herschieben, dann blättern Sie halt einfach 'mal das Handbuch so durch oder noch besser, Sie starten Edward an und probieren alles aus. Einige Besonderheiten dürften auch Ihnen auffallen.

I. Installation von Edward auf der Festplatte

Um das System auf Ihrer Festplatte zu installieren, sollten Sie sich zunächst ein geeignetes Verzeichnis anlegen, beispielsweise mit dem Namen „WORK:EDWARD". Alles weitere erfahren Sie aus der „Readme"-Datei auf der Diskette. Da sich auch nach der Drucklegung dieses Handbuches noch Änderungen ergeben können, wählten wir diesen Weg.

Hinweis: Edwardwird beim Installieren des MaxonC++-Compiler-Systems automatisch mitinstalliert.

II. Aufruf von Edward

1) Start von der Workbench

2) Start aus dem CLI

Je nachdem, was Sie bevorzugen, läßt sich Edward sowohl aus dem CLI als auch von der Workbench aus starten. Für die fortgeschritteneren Anwendern ist hierbei vor allem interessant, welche Parameter schon beim Start an Edward übergeben werden können.

Hinweis: In Zusammenarbeit mit dem C++-Compiler muß der Editor nicht extra gestartet werden. Dies übernimmt der Compiler für Sie. Lesen Sie die beiden folgenden Abschnitte, wenn SieEDWARD auch unabhängig von MaxonC++ anwenden möchten.

1) Start von der Workbench

Am einfachsten starten Sie Edward durch Doppelklick auf sein Icon. Dieses befindet sich nach einer normalen Installation im Verzeichnis „Edward:" auf der von Ihnen gewählten Partition Ihrer Festplatte oder auf der Diskette.

Sie können Edward ebenfalls starten, wenn Sie auf das Icon einer mit Edward bearbeiteten Datei geklickt haben.

☞ Wie Sie Icons zu einer Datei erzeugen lassen können und was es dabei zu beachten gilt, lesen Sie im Kapitel „Globale Einstellungen", insbesondere sollten Sie hier die Bemerkungen zum „Icon Tool" beachten.

Wenn Sie mit gedrückter Shift-Taste die Icons verschiedener Dateien anwählen und daraufhin Edward durch Doppelklick starten (Shift-Taste gedrückt lassen), so lädt Edward automatisch die von Ihnen angewählten Dateien ein. Angezeigt wird Ihnen allerdings nur die letzte geladene Datei.

☞ Zum gleichzeitigen Bearbeiten verschiedener Dateien lesen Sie mehr im Kapitel IV „Text".

2) Start aus dem CLI

Normalerweise starten Sie Edward von CLI aus durch einfache Eingabe von „Edward". Falls Ihnen das zu lang ist, so benennen Sie Edward einfach um oder Sie benutzen den „Alias"-Befehl Ihrer Shell (Mehr dazu lesen Sie in der von Commodore ausgelieferten Dokumentation). Sie können selbstverständlich Namen von Dateien angeben, die nach dem Start von Edward automatisch geladen werden sollen.
Nach dem Start von Edward aus dem CLI erscheint die Meldung „Edward Process successfully detached". Das bedeutet, daß Edward von nun an unabhängig vom CLI als eigener Prozeß läuft. Sie können weiter mit dem CLI arbeiten, ohne daß dieser blockiert ist und können den CLI auch jederzeit beenden.

III. Eingabe und Bearbeitung von Texten

1) Die Status-Anzeige

2) BACKSPACE und DELETE

3) UNDO-Funktion

4) RETURN, ENTER und TAB

5) Die Cursor-Tasten und der Ziffernblock

6) Der Slider

7) Die Maus

Zu Beginn erläutere ich kurz die Bedeutung der Status-Anzeige, danach werde ich Ihnen die voreingestellte Tastaturbelegung erklären, so daß Sie alle üblichen Operationen bei der Eingabe eines Textes vornehmen können. Desweiteren werde ich noch zusätzliche Möglichkeiten vorstellen, wie Sie die Cursor-Position im Text verändern können.

1) Die Status-Anzeige

Wichtige Informationen über den gerade bearbeiteten Text erhalten Sie auf einen Blick, wenn Sie die Titelzeile eines momentan aktiven Text-Fensters von Edward betrachten. Ganz links erscheint dort immer der Name des gerade bearbeiteten Textes. Anschließend folgen zwei Zahlenangaben, die erste gibt die aktuelle Zeile an, die zweite die Spaltenposition innerhalb dieser Zeile.
Anschließend folgen vier durch Komma getrennte Buchstaben. Diese geben an, ob bestimmte Optionen für den aktuellen Text aktiv sind, dann ist der entsprechende Buchstabe groß, oder nicht (der Buchstabe ist dann logischerweise klein).
Das „c" ganz links gibt an, ob der Text seit dem Einladen oder nach dem letzten Abspeichern verändert wurde (großes „C" falls dies der Fall ist).
Ein großes „W" bedeutet, daß der Text im „Word-Wrap"-Modus bearbeitet wird.
Ein „O" gibt an, daß nun der „OverWrite"-Modus gilt und ein großes „L" bedeutet, daß der Text momentan ge-"locked" ist. Ge-"locked" bedeutet, der Text kann momentan nicht verändert werden. Dies ist zum Beispiel der Fall, wenn Sie den entsprechenden Text drucken lassen.

☞ Mehr zu den oben erwähnten Modi lesen Sie in den Kapiteln IX. „Das Edit-Prefs Formular“ und X. „Word-Wrap“.

2) BACKSPACE und DELETE

Die Tasten BACKSPACE und DELETE sind wie üblich belegt. Mit CTRL-DELETE können Sie die komplette aktuelle Zeile löschen.

3) UndoLine-Funktion

Edward bietet die Möglichkeit das Löschen einer Zeile oder Änderungen einer Zeile rückgängig zu machen, falls Sie nicht schon eine weitere Zeile verändert haben.

- Geben Sie zum Test folgendes ein : „Der dumme Detlev demoliert die Damentoilette“. Fahren Sie nun mit dem Cursor auf „Detlev“ und ändern Sie den Namen in „Dödel“. Drücken Sie nun CTRL-U. Daraufhin erscheint statt „Dödel“ wieder „Detlev“.
- Löschen Sie nun die komplette Zeile mit CTRL-DELETE. Mit CTRL-U erhalten Sie die komplett gelöschte Zeile wieder zurück.

4) RETURN, ENTER und TAB

Falls Sie RETURN drücken und der Cursor befindet sich innerhalb einer Zeile, so wird die Zeile an dieser Position aufgespalten. Ähnliches gilt für ENTER. Drücken Sie RETURN oder ENTER am Anfang einer Zeile, so wird vor dieser eine leere Zeile eingefügt. Drücken Sie RETURN oder ENTER am Ende einer Zeile, so wird nach dieser Zeile eine leere Zeile eingefügt. Der Unterschied zwischen RETURN und ENTER ist, daß ENTER die neu erzeugte Zeile entsprechend der vorigen Zeile einrückt. Dies ist ein besonders für Programmierer nützliches Verhalten, das das strukturierte Programmieren unterstützt.
Mit der TAB-Taste fügen Sie ein Zeichen in den Text ein, daß den Platz von 3 Leerzeichen einnimmt. Dies dient ebenfalls der besseren Strukturierung von Texten und Programmen.

☞ Die Veränderung der TAB-Weite wird im Kapitel IX. über das Edit-Prefs-Formular erklärt.

Um die oben erklärten Funktionen von TAB und ENTER zu testen, können Sie folgendes ausprobieren.

- Drücken Sie dreimal die TAB-Taste. Wie Sie an der rechten Zahl in der Titel-Leiste des Fensters sehen, befindet sich der Cursor nun an der Position 10 im Text. Geben Sie nun „for(i=10; i; i--)“ oder was Ihnen beliebt ein.
- Drücken Sie nun die ENTER-Taste. Sie befinden sich nun in einer neuen Zeile und der Cursor befindet sich wieder auf Position 10, also auf derselben Einrückungsstufe wie die vorige Zeile.

5) Die Cursor-Tasten und der Ziffernblock

Wenn Sie die CURSOR-Tasten ganz normal drücken, also nicht in Kombination mit ALT, CTRL oder SHIFT, so bewegen Sie den Cursor wie gewohnt. Dies ist ebenfalls mit den Tasten des Ziffernblock möglich. Diese müssen Sie allerdings in Kombination mit der ALT-Taste drücken (ALT- 8, 4, 6, 2).

In Verbindung mit ALT bewegen Sie sich mit CURSOR-Left und CURSOR-Right um ein Wort vorwärts oder rückwärts. Mit CURSOR-Up und CURSOR-Down geht es jeweils vier Zeilen nach oben bzw. unten.

Mit SHIFT bewegen Sie den Cursor mit CURSOR-Left und CURSOR-Right an den Anfang bzw. das Ende der Zeile. In Verbindung mit CURSOR-Up und CURSOR-Down bewegen Sie sich jeweils um fast eine Bildschirmseite nach oben bzw. unten.
Mit ALT-PgUp (ALT-9) und ALT-PgDn(ALT-3) des Ziffernblocks bewegen Sie den Cursor auf dieselbe Weise.

Mit ALT-Home (ALT-7) und ALT-End (ALT-1) des Ziffernblock springen Sie an den Anfang bzw. das Ende des Textes.

Benutzen Sie die Tasten „8“ und „2“ des Ziffernblocks in Kombination mit der linken AMIGA-Taste, so können Sie den Text um jeweils eine Zeile nach oben oder unten scrollen. Dabei bleibt der Cursor an seiner Position, und der Text verschiebt sich unter ihm in beide Richtungen.

6) Der Schieberegler

Sie können die Position des Cursors im Text ebenfalls mit dem Slider an der rechten Seite des Textfenster bestimmen. Wenn Sie vor oder hinter den Slider klicken, bewegen Sie den Cursor immer um eine Seite vor oder zurück.

7) Die Maus

Es ist zusätzlich möglich, den Cursor mit der Maus an eine bestimmte Position zu bewegen. Dazu bewegen Sie einfach den Mauspfeil auf die gewünschte Position und drücken die linke Maustaste. Solange Sie die linke Maustaste gedrückt halten, folgt der Cursor dem Mauspfeil. Bewegen Sie sich an den oberen oder unteren Rand des Fensters, so scrollt der Text in die entsprechende Richtung. Die Geschwindigkeit, mit welcher dieser Vorgang abläuft, hängt davon ab, ob sich der Mauspfeil ganz rechts - dies entspricht einer sehr langsamen Bewegung - oder ganz links - dies entspricht einer sehr schnellen Bewegung, - befindet.

IV. Text

1) Erzeugen von Texten

2) Wahl eines bestimmen Textes

3) Freigeben eines Textes

4) Löschen des aktuellen Textes

5) Drucken des aktuellen Textes

Ein Text besteht einfach aus einer oder mehreren Zeilen, die Sie eingeben oder als Datei von Diskette oder Festplatte geladen haben.

☞ Das Laden von Dateien (=Texten) finden Sie im Kapitel V. „Dateioperationen." Mit Edward können Sie fast beliebig viele Texte im Speicher halten, zwischen diesen umschalten und (natürlich :-)) die Texte bearbeiten.

1) Erzeugen von Texten

- Starten Sie Edward.

Edward hat automatisch alles Nötige unternommen, damit Sie sofort einen Text oder ein Programm eingeben können.

- Tippen Sie nun „Mein 1. Text mit Edward" ein.

- Wir werden jetzt Edward mitteilen, daß wir einen weiteren Text bearbeiten wollen. Dazu wählen Sie aus dem „Text"-Menü den Eintrag „Neu". Das Fenster von Edward ist daraufhin wieder leer. Das ist richtig, denn Sie befinden sich jetzt im zweiten Text, für den Sie eben Speicher reserviert haben.

- Tippen Sie nun „Mein 2. Text mit Edward" ein.

Mit dem oben beschriebenen Menüpunkt können Sie jederzeit Edward mitteilen, daß Sie einen weiteren Text bearbeiten wollen. Verfahren Sie wie oben beschrieben und erzeugen Sie einen dritten Text und geben daraufhin „Mein 3. Text mit Edward" ein.

2) Wahl eines bestimmten Textes

Um wieder zurück zum ersten Text zu gelangen, gibt es verschiedene Möglichkeiten.

- Wählen Sie den Eintrag „Vorheriger" aus dem Text-Menü. Edward zeigt Ihnen nun die Zeile „Mein 2. Text mit Edward" an. Mit „Vorheriger" wählen Sie immer den vorigen Text aus.

- Wählen Sie erneut den Eintrag „Vorheriger" aus dem Text-Menü. Edward zeigt Ihnen nun die Zeile „Mein 1. Text mit Edward" an. Wir befinden uns also jetzt, wie gewünscht, wieder im ersten Text.

- Wählen Sie noch einmal den Eintrag „Vorheriger" aus dem Text-Menü. Edward zeigt Ihnen die Zeile „Mein 3. Text mit Edward" an. Vor dem ersten Text kann natürlich kein weiterer liegen. Darum wählt Edward in diesem Fall den letzten Text. Sie können also mit „Vorheriger" die Texte der Reihe nach anwählen.

Genauso können Sie durch Wählen von „Nächster" im „Text"-Menü den folgenden Text anwählen. Haben Sie gerade den letzten Text bearbeitet, so wählt Edward in diesem Fall den ersten Text.

- Eine direkte Wahl des Textes ist ebenfalls möglich. Möchten Sie den zweiten Text bearbeiten, dann wählen Sie einfach „Text 2" aus dem „Text"-Menü.

3) Freigeben eines Textes

Nach der Bearbeitung eines Textes (und dem Abspeichern desselben) können Sie Edward mitteilen, daß Sie diesen Text freigeben möchten.

- Wählen Sie „Löschen" aus dem „Text"-Menü. Edward zeigt Ihnen daraufhin die Zeile „Mein 3. Text mit Edward" an. Der zweite Text wurde von Edward freigegeben. Nach der Freigabe eines Textes versucht Edward, den folgenden zu wählen. Haben Sie den letzten Text freigegeben, dann wählt Edward den vorigen Text für die weitere Bearbeitung aus.

☛ Hinweis:
Sollten Sie aus Versehen die Freigabe eines Textes eingeleitet haben, der noch nicht abgespeichert wurde, so erscheint ein Dialogfenster mit der Frage, ob Sie diesen Text „vergessen" wollen. Geben Sie „Nein" an, so wird die Operation abgebrochen, wählen Sie „Ja", dann gehen alle Änderungen, die an diesem Text vorgenommen wurden, verloren!

4) Löschen des aktuellen Textes

Mit dem Eintrag „Entfernen" aus dem „Text"-Menü können Sie alle Zeilen des aktuellen Textes löschen und danach einen neuen Text eingeben. Der Text wird dadurch aber nicht freigegeben.

☛ Hinweis:
Sollten Sie aus Versehen die Löschung eines Textes eingeleitet haben, der noch nicht abgespeichert wurde, so erscheint ein Dialogfenster mit der Frage, ob Sie diesen Text „vergessen" wollen. Geben Sie „Nein" an, so wird die Operation abgebrochen, wählen Sie „Ja", dann gehen alle Änderungen, die an diesem Text vorgenommen wurden, verloren!

5) Drucken eines Textes

Mit dem Eintrag „Drucken" aus dem „Projekt"-Menü teilen Sie Edward mit, daß er den Text, den Sie momentan bearbeiten, ausdrucken soll. Während Edward diesen Text druckt, können Sie weiterhin andere Texte bearbeiten und desweiteren alle Funktionen von Edward benutzen, die den aktuellen Text nicht verändern. Sie können also, während Edward Ihren Text druckt, weiterhin in diesem den Cursor positionieren oder z.B. Blöcke markieren und kopieren. Das Einfügen oder Ausschneiden von Blöcken ist dagegen nicht möglich.
Während des Druckvorgangs erscheint ein kleines Fenster. Wenn Sie den Druckvorgang unterbrechen wollen, so schließen Sie einfach dieses Fenster.

☞ Das Format, in welchem eine Textseite gedruckt werden soll, können Sie auf vielfältige Weise variieren. Lesen Sie dazu mehr im Kapitel XVI. „Globale Einstellungen".

☛ Wichtig:
Edward gibt den Text an den Drucker aus, den Sie mit den Preferences der Workbench gewählt haben. Sollten Sie hiermit nicht klarkommen, so schauen Sie im Handbuch zur Workbench nach, das zu jedem Amiga mitgeliefert wird.

V. Dateioperationen

1) Speichern von Texten

2) Laden von Texten

Hinter dem hochtrabenden Titel „Dateioperationen" verbergen sich Funktionen zum Abspeichern und Laden von Texten.

1) Speichern von Texten

Nachdem Sie im ersten Kapitel gelernt haben, wie man einen Text eingibt und bearbeitet, zeige ich Ihnen nun, wie man einen solchen Text auf Diskette oder Festplatte abspeichert und wie man den Text von dort zur Bearbeitung wieder in den Editor bekommt.

- Geben Sie nun einen beliebigen Text ein, z.B.: „Alles Seiende hat sein Sein im Nicht-Sein seines Gegensatzes."

- Wählen Sie aus dem „Projekt"-Menü den Eintrag „Speichern als...". Es erscheint daraufhin ein Dateiauswahlfenster, in dessen Titelzeile „Text speichern" steht. Speichern Sie den Text unter dem Namen „Test.TXT" ab.

Normalerweise wird beim Abspeichern eines Textes für diesen ein Pictogramm erzeugt, so daß er auch auf der Workbench sichtbar wird. Desweiteren werden einige Zusatzinformationen, die den Text betreffen mitabgespeichert (unter anderem die Tabulatorweite, Textfalten, etc.). Besonders interessant ist hierbei, daß Edward auch die aktuelle Cursorposition speichert. Das bedeutet, wenn Sie einen Text erneut einladen, dann befindet sich der Cursor an derselben Stelle, an der Sie zuletzt gearbeitet haben.

☞ Das Erzeugen des Pictogrammes und das Speichern der Zusatzinformationen können Sie natürlich auch abstellen. Lesen Sie hierzu das Kapitel XVI. „Globale Einstellungen".

☛ Hinweis:
Unter Umständen erscheint ein „System Request" mit dem Hinweis, das Ihre Diskette schreibgeschützt ist. Sorgen Sie dann dafür, das der Schreibschutz entfernt wird oder nehmen Sie eine andere Diskette.

☛ Wichtig:
Die Bedeutung des Abspeicherns kann gar nicht oft genug betont werden. Alle Veränderungen, die Sie seit dem letzten Abspeichern vorgenommen haben, sind verloren, wenn ein System-Absturz auftritt, der Strom ausfällt oder sonstige unvorhersehbare Ereignisse eintreten. Sie können durch eine spezielle Einstellung dafür sorgen, daß Edward in bestimmten Zeitabschnitten Ihre Texte automatisch abspeichert.

☞ Prefs/Autosave

2) Laden von Texten

Nachdem Sie nun einen Text abgespeichert haben, zeige ich Ihnen im Folgenden wie Sie diesen Text wieder einladen.

- Wählen Sie aus dem „Text"-Menü den Eintrag „Entfernen". Das Textfenster ist nun leer. Alle Textzeilen wurden freigegeben. Dies ist nur für unser Beispiel notwendig.

- Wählen Sie nun den Eintrag „Öffnen" aus dem „Projekt"-Menü. Es erscheint wie vorher beim Abspeichern der Dateiauswahlfenster. In diesem Fall steht in seiner Titelzeile aber „Text laden". Laden Sie den Text „Test.TXT" wieder ein.

☛ Hinweis:
Wenn Sie einen Text einladen, wird dieser über den Inhalt Ihres aktuellen Textes geladen. Wenn Sie dies vermeiden wollen, so sollten Sie zuerst einen neuen Text erzeugen, bevor Sie „Öffnen" wählen. Sollte der aktuelle von Ihnen bearbeitete Text noch nicht gespeichert worden sein, so erscheint ein Dialog-Fenster, das Sie auf diesen Umstand aufmerksam macht und mit dem Sie das Überladen dieses Textes verhindern können.

VI. Blockoperationen

1) Cut, Copy, Paste

2) Spaltenblöcke

3) Laden, Speichern und Drucken

4) Einrücken

5) Umwandeln in Groß- bzw. Kleinbuchstaben

Größere Teile eines Textes umgruppieren oder vervielfältigen zu können, ist eine der wichtigsten Fähigkeiten eines Editors. Edward bietet Ihnen auch auf diesem Gebiet alles Nötige und einiges darüber hinaus.
Ein besonders markierter Bereich eines Textes wird im allgemeinen als „Block" bezeichnet. Daraus leitet sich auch der Name dieses Kapitels ab.

1) Cut, Copy, Paste

Hinter den Begriffen „Ausschneiden", „Kopieren" und „Einfügen" verbergen sich die elementaren Blockoperationen.
Mit „Ausschneiden" schneiden Sie einen markierten Block aus Ihrem Text aus und speichern ihn in einem Zwischenspeicher ab. „Kopieren" kopiert den Block nur in den Zwischenspeicher, ohne ihn auszuschneiden und „Einfügen" fügt den Block ab der aktuellen Cursor-Position in den Text ein.

Um eine Blockoperation ausführen zu können, müssen Sie zuerst einmal einen Block markieren. Dies geschieht indem Sie „Markieren" aus dem „Block"-Menü wählen. Wenn Sie nun den Cursor bewegen, so werden Sie feststellen, daß der gesamte Textbereich zwischen der Position an der sich der Cursor momentan befindet und dem Ort, an dem sich der Cursor befand, bevor Sie „Markieren" gewählt haben, invertiert dargestellt wird. Dieser invertiert dargestellte Bereich ist der sogenannte markierte „Block". Alle Blockoperationen wirken sich auf diesen Bereich aus.
Wenn Sie erneut „Markieren" wählen, wird der gesamte Text wieder normal dargestellt. Der markierte Block wurde damit vergessen.

- Für die folgenden Schritte sollten Sie nun einen etwas längeren Text einladen.

- Positionieren Sie nun den Cursor in die Mitte irgendeiner Zeile am Anfang des Textes und wählen Sie danach „Markieren" aus dem „Block"-Menü.

- Fahren Sie nun mit dem Cursor in irgendeine Zeile, die weiter unten liegt. Sie sehen nun einen invertiert dargestellen Bereich.

- Wählen Sie „Ausschneident" aus dem „Block"-Menü. Der markierte Bereich verschwindet daraufhin. Der Cursor befindet sich nun auf derselben Position, an der er sich befand, bevor Sie „Markieren" gewählt hatten. Rechts vom Cursor befindet sich der Rest der Zeile, auf der sich der Cursor befand, bevor Sie „Ausschneiden" gewählt hatten.

Um nun die ursprüngliche Gestalt des Textes wiederherzustellen, können Sie den ausgeschnittenen Bereich wieder in den Text einfügen. Dazu dürfen Sie den Cursor nicht bewegen.

- Wählen Sie „Einfügen" aus dem „Block"-Menü. Der Cursor befindet sich nun am Ende des eingefügten Bereiches. Ihr Text hat wieder seine ursprüngliche Gestalt.

Sie können jederzeit an anderen Positionen im Text eine Kopie des ausgeschnittenen Bereiches einfügen, indem Sie einfach „Paste" anwählen.
Wenn Sie einen bestimmten Textbereich nur kopieren wollen, so markieren Sie den Bereich, wählen „Kopieren", positionieren den Cursor an die gewünschte Stelle und wählen danach „Einfügen".

☛ Zum Zwischenspeichern von Textausschnitten bedient sich Edward des Zwischenspeichers. Dies ist eine genormte Ablage, der sich alle Programme, die für den Amiga entwickelt werden, bedienen sollten. Falls Sie zum Beispiel ein Desktop-Publishing Programm besitzen, aber nicht auf die komfortablen Editier-Möglichkeiten Edwards verzichten wollen, so können Sie sich des Zwischenspeichers bedienen, um Texte zwischen Edward und dem DTP-Programm auszutauschen. Dabei ist natürlich Voraussetzung, daß das DTP-Programm ebenfalls den Zischenspeicher unterstützt.

2) Spaltenblöcke

Eine Besonderheit im Umgang mit Blöcken bietet Ihnen Edward mit den Spaltenblöcken. Haben Sie zum Beispiel eine Tabelle mit mehreren Spalten eingegeben und Sie

wollen nun eine der Spalten kopieren oder vertauschen, dann bewerkstelligen Sie dies unter Zuhilfenahme von Edwards Spaltenblöcken.

- Geben Sie folgende Tabelle ein

Film	Bewertung	Hauptdarsteller	!Test
Alien I	1-	Sigourney Weaver	!
Alien II	1	Sigourney Weaver	!
Terminator II	1++	Arnold Schwarzenegger	!
Homo Faber	1	Sam Shephard	!
Gefährliche Liebschaften	1+	John Malkovic,	!
		Glenn Close,	!
		Michelle Pfeiffer	!
The Fabulous Baker Boys	1	Michelle Pfeiffer,	!
		Jeff & Beau Bridges	!
Das Russlandhaus	2	Sean Connery,	!
		Michelle Pfeiffer	!
Harry & Sally	1	Meg Ryan	!
Moon on Earth	1	Wynona Ryder	!
Fessle mich!	1	-	!
Das Wunderkind Tate	2+	Jodie Foster	!
König der Fischer	2+	Robin Williams	!
Hook	6	Dustin Hoffmann,	!
		Robin William,	!
		Julia Roberts	!

Sie wollen nun die Spalte „Bewertung“ mit der Spalte „Hauptdarsteller“ vertauschen.

- Fahren Sie mit dem Cursor auf das B von „Bewertung“. Wählen Sie dann „vertikal markieren“ aus dem „Block“-Menü.

- Bewegen Sie nun den Cursor vor das R von „Robin Williams“ in der letzten Zeile, Spalte „Hauptdarsteller“. Wählen Sie anschließend „Ausschneiden“ aus dem „Block“-Menü. Die Spalte „Bewertung“ ist nun verschwunden, „Hauptdarsteller“ befindet sich an deren Stelle.

- Positionieren Sie den Cursor auf das Ausrufungszeichen in der ersten Zeile der Spalte „Test“. Wählen Sie nun „vertikal einfügen“ aus dem „Block“-Menü.

Daraufhin erscheint die Spalte „Bewertung“ wieder. Die Vertauschung hat funktioniert.

3) Laden, Speichern und Drucken

Sie können einen mit „Ausschneiden" oder „Kopieren" in die Zwischenablage kopierten Block auf einfache Weise als Datei abspeichern. Dazu wählen Sie einfach „Block speichern" aus dem „Bloxk"-Menü. Es erscheint der Dateiauswahlfenster und Sie können einen Namen für den Block angeben.
Wollen Sie einen so abgespeicherten Block wieder in Ihren Text einfügen, so wählen Sie „Block einlesen" aus dem „Block"-Menü. Mit dem Dateiauswahlfenster geben Sie eine Datei an. Daraufhin wird die angebene Datei in den Zwischenspeicher geladen. Sie können sie dann jederzeit mit „Einfügen" an einer beliebigen Stelle im Text einfügen.

Mit dem Menüpunkt „Block drucken" aus dem „Block"-Menü können Sie einen in die Zwischenablage gespeicherten Block ausdrucken, allerdings ohne die Ausgabe besonders zu formatieren, wie es beim Ausdruck eines ganzen Textes geschieht.

4) Einrücken

Zur Strukturierung von Texten, insbesondere von Programmen, dient das Einrücken von Textabschnitten, die damit als zusammengehöriger Abschnitt hervorgehoben werden sollen. Zur Unterstützung dieser Maßnahmen bietet Ihnen Edward die Möglichkeit, einen markierten Bereich von Zeilen einzurücken.

- Geben Sie folgendes Programmfragment ein :

```
for( i=1000; i>0; i-- )
{
  c=c0*c0+c1*c1-r;
  c++;
  r++;
}
```

Um den Bereich zwischen der öffnenden und der schliessenden geschweiften Klammer ("{","}") besser hervorzuheben, können Sie ihn einrücken.

- Bewegen Sie dazu den Cursor auf die Zeile nach der öffnenden geschweiften Klammer und wählen Sie „Mark" aus dem „Edit"-Menü.

- Fahren Sie nun mit dem Cursor an das Ende der Zeile vor der schliessenden geschweiften Klammer.

Mit CTRL-CursorLeft und CTRL-CursorRight können Sie nun den markierten Bereich um jeweils einen TAB-Schritt nach rechts oder links verschieben.

- Drücken Sie CTRL-CursorRight. Damit haben Sie den Block um eine TAB-Position nach rechts verschoben. Probieren Sie ruhig ein bißchen die Möglichkeiten aus, die sich mit dem oben beschriebenen Verfahren ergeben.

5) Umwandeln in Groß- bzw. Kleinbuchstaben

Mit zwei weiteren Funktionen können Sie alle Buchstaben eines markierten Blocks in Groß- bzw. Kleinbuchstaben wandeln lassen.

- Geben Sie „Alles Seiende entsteht ohne Grund, setzt sich aus Schwäche fort und stirbt durch Zufall." oder etwas was Ihnen mehr zusagt, ein.
- Markieren Sie einen Block, der vom ersten Buchstaben von „Seiende" bis zum letzten Buchstaben von „aus" reicht.
- Wählen Sie nun „Großbuchstaben" aus dem „Extras"-Menü. Augenblicklich werden alle markierten Buchstaben in Großbuchstaben umgewandelt.

Wenn Sie „Kleinbuchstaben" aus dem „Extras"-Menü wählen, so werden entsprechend alle Buchstaben des markierten Blocks in Kleinbuchstaben gewandelt.

VII. Suchen und Ersetzen

1) Das „Find"-Formular

2) Das „Find & Replace"-Formular

3) Struktur-Check (Klammer-Check)

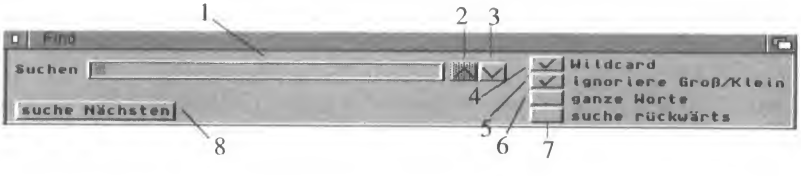
4) Compiler Error-Files interpretieren und Fehler anspringen

5) Auf bestimmte Zeilen positionieren

Suchen und Ersetzen von Zeichenfolgen in einem Text gehören zum Standardrepertoire eines Editors. Hier läßt Sie Edward natürlich nicht im Stich.
Um Ihnen das Arbeiten beim Suchen und Ersetzen zu erleichtern, bietet Edward zwei Extra-Fenster mit diversen Einstellmöglichkeiten an, über welche Sie die Arbeiten beim Suchen und Ersetzen steuern. Da Sie in diesen Fenstern gewisse Eintragungen machen können, sprechen wir im folgenden von sogenannten „Formularen".

1) Das „Find"-Formular

Zum Suchen eines bestimmten Begriffes bedienen Sie sich des „Find"-Formulars. Dieses erscheint, wenn Sie im Menü „Suchen" den Menüpunkt „Suchen ..." wählen.



Wir werden nun gemeinsam die Funktionen des „Find"-Formulars durchsprechen.

1) In diesem Bereich geben Sie die zu suchende Zeichenfolge ein. Nach dem Öffnen eines Find-Formulars ist 1) automatisch aktiv und zur Eingabe bereit. Nach dem Aufruf des „Find"-Formulars ist er sofort zur Eingabe bereit. Wenn Sie die

Eingabe mit RETURN abschliessen, so verschwindet das „Find"-Formular gleich darauf und der Cursor wird auf die erste Stelle im Text, wo diese Zeichenfolge gefunden werden konnte positioniert.
Für die Suche einer Zeichenfolge gilt es dabei Folgendes zu beachten:
Gesucht wird immer von der aktuellen Cursorposition aus. Normalerweise wird von dieser ausgehend dann in Richtung auf das Ende des Textes hin gesucht. Konnte dabei keine Zeichenfolge im Text gefunden werden, die der gewünschten entspricht, so wird die Suchrichtung umgekehrt. Jeder weitere Suchvorgang geht dann von der Cursorposition aus zum Beginn des Textes hin. Wird dort nichts mehr gefunden, so kehrt sich die Richtung erneut um.

7) Mit 7) können Sie die Suchrichtung jederzeit umkehren.

4) Ist 4) angewählt, so können beim Suchen sogenannte WildCards eingesetzt werden. Als sog. Jokerzeichen dient dabei der Stern „*". An jeder Stelle an der sich in der zu suchenden Zeichenfolge ein Stern befindet darf beim Vergleich mit einer anderen Zeichenfolge eine beliebige Teilfolge stehen.
Geben sie zum Beispiel „Co*m*e" als zu suchende Zeichenfolge ein, so wird beim Vergleich unter anderem „Commodore" oder „Commode" als gefunden erkannt.

5) Wenn Sie 5) anwählen, so wird beim Suchen nicht zwischen Groß- und Kleinbuchstaben unterschieden.

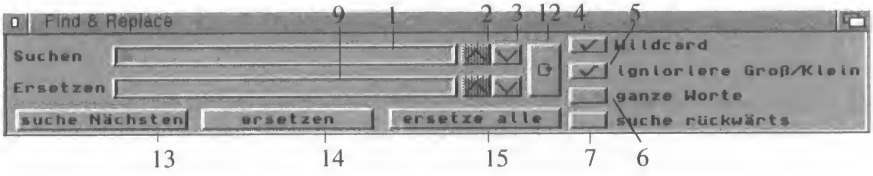
6) Wählen Sie 6), so wird nur nach vollständigen Wörtern gesucht. Wenn Sie also die Zeichenfolge „Kohl" suchen und Sie haben 6) gewählt, so gilt „Kohlsuppe" oder „Kohldampf" als nicht gefunden. Nur „Kohl „ wird gefunden und der Cursor entsprechend positioniert.

2), 3) Eine Besonderheit bietet Edward Ihnen mit 2) und 3). Sie können insgesamt 10 verschiedene Suchmuster eingeben. Mit den Pfeilen können Sie zwischen Ihnen umschalten.

8) Mit 8) positionieren Sie den Cursor automatisch auf das nächste Vorkommen der in 1) angebenen Zeichenkette.

2) Das „Find & Replace"-Formular

Mit dem „Find & Replace"-Formular können Sie auf komfortable Weise Zeichenfolgen im Text suchen und durch andere Zeichenfolgen ersetzen lassen.



Die Bedienung des „Find & Replace"-Formular ist mit der des „Find"-Formular weitgehend identisch. Dies gilt insbesondere für die durch 4) - 8) wählbaren Optionen.

1) In 1) geben Sie die zu suchende Zeichenfolge an, in 9) die Zeichenfolge durch welche 1) ersetzt werden soll. Durch das Öffnen des „Find & Replace"-Formulars wird automatisch 1) aktiviert. Nach der Eingabe in 1) wird automatisch 9) aktiviert.

2),3) 10),11) Mit 2) und 3) können Sie zehn verschieden Suchmuster anwählen, mit 10) und 11) analog zehn verschieden Zeichenfolgen durch die diese ersetzt werden sollen.

12) Als weitere Komfortsteigerung können Sie durch 12) die Such- und Ersetzungszeichenfolgen miteinander vertauschen. Falls Sie also aus Versehen zum Beispiel „Hund" suchen und durch „Katze" ersetzen ließen so können sie durch anklicken von 12) und erneutes Ersetzen das Ganze rückgängig machen.

13) Mit 13) positionieren Sie den Cursor automatisch auf das nächste Vorkommen der in 1) angebenen Zeichenkette.

9),14) Klicken Sie auf 14) dann wird von der aktuellen Cursor-Position aus die in 1) angebebene Zeichenfolge gesucht und, falls gefunden, durch die in 9) angebenene Zeichenfolge ersetzt. Danach wird automatisch das nächste Vorkommen von 1) gesucht und der Cursor darauf positioniert.

15) Mit 15) werden im gesamten Text automatisch alle Vorkommen von 1) durch 9) ersetzt. Anschließend wird der Cursor auf den Beginn des Textes positioniert.

3) Struktur-Check (Klammer-Check)

Für Programmierer (insbesondere diejenigen, die die Sprache „C" benutzen) kann der Strukur-Check sehr hilfreich sein. Mit diesem läßt sich auf einfache Weise prüfen, ob für jede öffnende Klammer in einem Ausdruck auch eine schließende Klammer vorhanden ist. Dabei werden nicht nur die gewohnten runden Klammern „(„ und „)" überprüft, sondern auf Wunsch auch „[" und „]" sowie „{" und „}".

- Geben Sie folgende Zeilen ein :

```
void main( void )
{
  int i;
  for( i=100; i>0; i-- )
  {
    if( i & 1 )
    { printf( "%d\n", i*i+((3*i)-10/i) );
    }
  }
}
```

- Positionieren Sie nun den Cursor auf die öffnende geschweifte Klammer in der zweiten Zeile.
- Nun wählen Sie „Strukturkontrolle“ aus dem „Suchen“-Menü. Der Cursor wird automatisch auf die korrespondierende schließende geschweifte Klammer in der letzten Zeile positioniert.
- Wenn Sie erneut „Strukturkontrolle“ wählen, dann wird der Cursor wieder auf die öffnende geschweifte Klammer in der zweiten Zeile gesetzt.

Die Funktion „Strukturkontrolle“ versucht immer den Cursor auf die zur Klammer unter dem Cursor korrespondierende öffnende oder schließende Klammer zu setzen. Sollte dies nicht möglich sein, so meldet Edward „Structures do not match!“. Somit können Sie bei stark verschachtelten Funktionsblöcken in Ihren Programmen auf einfache Weise überprüfen, ob Sie zu jeder öffnenden auch eine schließende Klammer gesetzt haben und Sie müssen nicht erst den Compiler bemühen, um solche leicht zu vermeidenden Syntax-Fehler zu entdecken.
Mit den runden Klammern in der sechsten Zeile des obigen Beispiels können Sie die Funktion „Match Bracket“ ausprobieren. Nehmen Sie ruhig einmal eine Klammer heraus und stellen Sie fest, wie Edward darauf reagiert.

4) Compiler Error-Datei interpretieren, Fehler anspringen

Eine Besonderheit von Edward ist, daß er die Fehlerdateien bestimmter Compiler auswerten kann. Sie müssen nur von Ihrem Compiler eine Fehlerdatei erzeugen lassen und diese mit Edward laden. Die dafür nötigen Funktionen können Sie zum Beispiel auf eine Taste oder ein Menü legen. Bei einem C-Compiler können Sie durch einfachen Aufruf von „CC >ErrorFile ...“ die Fehlermeldungen in die Datei „ErrorFile“ umleiten. Wenn Sie die Fehlerdatei geladen haben, so positionieren Sie den Cursor auf eine Zeile in der die Zeilennummer eines Fehlers steht. Danach wählen Sie „Gehe zu Fehler“. Daraufhin überprüft Edward, ob die Datei, zu der die Fehlerbeschreibung gehört, sich schon im Speicher befindet. Ist dies nicht der Fall, so wird die entsprechende Datei nachgeladen. Daraufhin wird der Cursor auf die Zeile positioniert, in welcher sich der angebene Fehler befindet.

☛ Wenn Sie in einer geladenen Quelldatei einige Veränderungen vornehmen und dabei Zeilen einfügen oder löschen, so kann „Gehe zu Fehler“ natürlich nicht mehr auf die richtige Zeile positionieren. Sie sollten daher immer die ungefähren Verschiebungen von Zeilen im Kopf behalten.

5) Auf bestimmte Zeilen positionieren

Mit „Gehe zur Zeile..“ öffnen Sie ein kleines Formular, in dem sofort ein Feld zur Eingabe einer Zahl bereit ist. Geben Sie die Nummer der gewünschten Zeile ein und drücken Sie „RETURN“. Daraufhin versucht Edward den Cursor auf die gewünschte Zeile zu positionieren.

Mit „Gehe zu m Anfang“ können Sie an den Anfang eines Textes mit „Gehe zum Ende“ an das Ende eines Textes gelangen.

VIII. Strukturieren von Texten, Lesezeichen

1) Erzeuge Falte, Zeige Falte, Entferne Falte, Zeige alle Falten, Schließe alle Falten

2) Auszeichnen von Zeilen

   Mark, UnMark

3) Cursor-Bewegung

4) Lesezeichen (Bookmarks)

Programme bestehen normalerweise aus Prozeduren, die in weitere kleine Einheiten strukturiert sind (oder zumindest sein sollten). Um ein Programm etwas übersichtlicher darstellen zu können, bietet Ihnen Edward das Konzept der Falten an. Desweiteren können Sie bestimmte Zeilen markieren. Mit den Cursor-Tasten lassen sich so bestimmte Strukturen einer Datei leicht anwählen. Ausserdem können wichtige Zeilen gemerkt und mit einem Tastendruck direkt angesprungen werden (Lesezeichen).

☛ Hinweis:
Die Strukturierung eines Textes wird in der zum Text gehörigen Pictogramm-Datei abgespeichert. Näheres hierzu erfahren Sie im Kapitel XVI. „Globale Einstellungen".

1) Erzeuge Falte, Zeige Falte, Entferne Falte, Zeige alle Falten, Schließe alle Falten

Hinter dem Konzept der Falten verbirgt sich eine sehr leistungsfähige Methode zur Strukturierung von Programm-Quelldateien.

- Geben Sie nun folgendes kleine Programm ein oder laden Sie die Quelldatei eines von Ihnen erstellten Programms ein.

```
void strnrcpy
(
  char *string0,
  char *string1,
  int n
)
{
  int Pos;

  Pos=strlen( string1 );
  if( Pos>n )
    Pos=Pos-n;
  else
    Pos=0;
  strcpy( string0, &string1[Pos] );
}

void strtolower
(
  register char *StringP
)
{
  register char c;
  while( (c=*StringP)!=EOS )
  {
    if( isalpha(c) )
      *StringP++=_tolower(c);
    else
      StringP++;
  }
}
```

- Bewegen Sie den Cursor auf die erste Zeile. Wählen Sie „Markieren“ aus dem „Block“-Menü. Positionieren Sie den Cursor nun unter die letzte Zeile der Prozedur „strnrcpy“.
- Jetzt wählen Sie „Erzeuge Falte“ aus dem „Falte“-Menü. Alle markierten Zeilen bis auf die erste verschwinden nun. Vor dieser erscheint das Zeichen „«“. Der Cursor wurde ebenfalls auf diese Zeile bewegt.

Damit haben Sie die erste Falte erzeugt. Alles, was vorher markiert war, steckt nun in der Falte. Nur die erste Zeile ist noch sichtbar. Ein besonderes Zeichen zeigt Ihnen den Beginn der Falte an.

- Wählen Sie nun „Zeige Falte". Daraufhin wird der Inhalt der Falte wieder sichtbar. Die Zeilen innerhalb der Falte beginnen mit dem Zeichen „|". Die letzte Zeile einer Falte wird durch das Zeichen „»" kenntlich gemacht.

- Wählen Sie erneut „Erzeuge Falte", dann wird die Falte wieder „eingeklappt" und die vorher angezeigten Zeilen verschwinden.

- Wenn Sie „Entferne Falte" anwählen, so verschwindet die Markierung der gefalteten ersten Zeile, und alle „innereren" Zeilen der Falte erscheinen wieder. Diesmal allerdings unmarkiert. Die Faltung wurde vollständig rückgängig gemacht.

☛ „Zeige Falte" und „Entferne Falte" arbeiten nur, wenn der Cursor auf die erste Zeile einer Falte positioniert wurde. „Erzeuge Falte" faltet einen markierten Block ein oder schließt eine mit „Zeige Falte" geöffnete Falte wieder.

- Falten Sie nun beide Prozeduren aus unserem Beispiel ein.
- Jetzt wählen Sie „Zeige alle Falten". Daraufhin erscheinen beide gefalteten Bereiche so, alls wäre für jeden die Funktion „Zeige Falte" aufgerufen worden.
- Durch Wahl von „Schließe alle Falten" wird das Ganze wieder rückgängig gemacht und nur die erste Zeile jeder Falte ist noch sichtbar.

Mit „Entferne alle Falten" entfernen Sie alle Falten aus Ihrem Text.

Selbstverständlich lassen sich die Falten ineinander schachteln. Das bedeutet, das Sie mehrere gefaltete Bereiche in einer übergreifenden Falte zusammenfassen können.

Mit „Zeige innere Falten" können Sie auf einen Schlag eine Falte mit all ihren inneren Falten öffnen. „Schließe innere Falten" macht dies rückgängig.
Mit „Entferne innere Falten" entfernen Sie eine Faltung inklusive all ihrer Unterfaltungen.

2) Auszeichnen von Zeilen

Bestimmte Zeilen eines Dokuments oder einer Quelldatei, die Sie mit Edward bearbeiten, können Sie als besonders wichtig kennzeichnen. Diese Zeilen können mit den in Abschnitt 3) erwähnten Methoden auf besonders einfache Weise angesprungen werden.

Eine Zeile wird durch die Tastenkombination „ALT-SHIFT-RETURN" ausgezeichnet. Die Zeile wird daraufhin mit einem führenden Ausrufungszeichen dargestellt. Wiederholen Sie diese Tastenkombination, so wird die Auszeichnung zurückgenommen, und das Ausrufungszeichen verschwindet.

3) Cursor-Bewegung

Mit zwei Tastenkombination können Sie gefalteten Zeilen sehr einfach anspringen. Mit „CTRL-CURSOR_UP" wählen Sie eine vor der aktuellen Zeile liegende Zeile an, die einen Faltenstart oder ein Faltenende darstellt. Ebenso können Sie mit „CTRL-CURSOR_DOWN" eine nach der aktuellen Zeile liegende Zeile anspringen.

Mit „ALT-SHIFT-CURSOR_UP" bzw. „ALT-SHIFT-CURSOR_DOWN" springen Sie eine vor bzw. nach der aktuellen Zeile liegende Zeile an, die Sie ausgezeichnet hatten.

4) Lesezeichen (Bookmarks)

Beim Arbeiten mit einem Text kommt es oft vor, daß man schnell verschiedene, weit auseinanderliegende Positionen im Text anspringen möchte, an denen man gleichzeitig arbeitet (z.B. ich an diesem Handbuch).
Eine Möglichkeit um dies zu erreichen wäre, die entsprechenden Zeilen auszuzeichnen und dann mit den Cursor-Tasten anzuspringen, dies hat jedoch den Nachteil, daß man unter Umständen mehrere Bewegungen im Text machen muß, um die gewünschte Stelle zu erreichen.
In diesem Fall gibt es aber eine weitere Besonderheit von Edward, die sogenannten „Lesezeichen". Mit der linken ALT-Taste und einer Zahlentaste des alphanumerischen Teils der Tastatur können Sie eine Zeile zwischenmerken. Wenn Sie nun an eine andere Stelle des Textes fahren und die linke AMIGA-Taste in Verbindung mit derselben Zahlentaste drücken, so springen Sie augenblicklich an die alte Stelle im Text.
Mit der Standard-Tastaturbelegung Edwards können Sie zehn Lesezeichen benutzen, theoretisch erlaubt Ihnen Edward jedoch 65536 Lesezeichen pro Text.

- Laden Sie eine Datei ein.
- Drücken Sie die linke ALT-Taste und die „1".

Es erscheint in der Titelzeile des Fensters die Meldung „Markierung gesetzt".

- Fahren Sie an eine beliebige andere Stelle des Textes.
- Drücken Sie nun die linke AMIGA-Taste und wieder die „1". Und schwupps sind Sie wieder an der alten Position im Text.

IX. Das Edit-Prefs Formular - Ändern der Voreinstellungen für die Texteingabe

Mit dem „Edit Prefs"-Formular, das Sie durch Wahl von „Edit Prefs..." aus dem „Text"-Menü erhalten, können Sie für jeden Text den Sie mit Edward bearbeiten diverse Einstellungen vornehmen.

1) Ist 1) aktiviert, dann wird bei der Standard-Tastaturbelegung auch nach Drücken der RETURN-Taste die neue Zeile entsprechend der alten eingerückt, wie es nach Drücken der ENTER-Taste der Fall wäre.

☞ Siehe auch Kapitel III. „Eingabe und Bearbeitung von Texten"

2) Durch Klick auf 2) schalten Sie den „Word-Wrap"-Modus an oder ab.

☞ Über den „Word-Wrap"-Modus erfahren Sie mehr im Kapitel X. „Word-Wrap".

3) Ist Overstrike angewählt, dann überschreiben Sie jeweils das Zeichen auf dem der Cursor steht durch das neugetippte. Normalerweise würde Edward das Zeichen einfügen, Overstrike überschreibt dagegen die alten Zeichen.

4) Ist diese Option aktiv, so werden alle eingegeben TABs in einer Zeile automatisch in entsprechend viele Leerzeichen umgewandelt.

5),6),7) Mit diesen drei Symbolen können Sie angeben, ob Tabs, Spaces und Zeilenenden einfach durch Leerräume dargestellt werden sollen (Normaleinstellung) oder besonders gekennzeichnet dargestellt werden sollen.

8) Dateien von anderen Computersystemen enthalten unter Umständen ein zusätzliches Zeichen um ein Zeilenende anzuzeigen. Durch Aktivieren dieser Option wird ein solches Zeichen automatisch beim Laden des Textes entfernt.

9) Tab Width gibt die Anzahl von Zeichen zwischen zwei Tabularpositionen an. Sie können diesen Abstand im Bereich 1 bis 20 wählen.

10) Mit Right Margin geben Sie die Position in einer Zeile an, ab der im „Word-Wrap"-Modus umgebrochen werden soll.

12) Wählen Sie USE, so werden die von Ihnen gewählten Einstellungen für den Text übernommen. Falls Sie „Right Margin" geändert haben, so wird der komplette Text dementsprechend neu umformatiert und der Cursor danach auf die erste Zeile des Textes gesetzt.

13) Verwirft die von Ihnen vorgenommenen Einstellungen.

X. „Word-Wrap“

Zum Erstellen von Fließtexten, wie sie unter anderem für DTP-Programme benötigt werden, können Sie Edward in dem „Word-Wrap“-Modus versetzen. „Word-Wrap“ bezeichnet ein spezielles Verhalten des Editors, bei dem eingegebene Zeilen beim Überschreiten einer bestimmten Länge umgebrochen werden. Ein Wort, das mitten auf der Umbruchposition liegt, wird automatisch in die nächste Zeile gezogen. Wird diese Zeile dadurch ebenfalls zu lang, so wird das Ganze fortgesetzt und zwar bis an das Ende eines Absatzes.

Ein Absatz wird dadurch gekennzeichnet, daß Sie nach der letzten Zeichenfolge des Absatzes „RETURN“ drücken. Im Gegensatz zum normalen Editier-Modus von Edward ist es im „Word-Wrap“-Modus nicht notwendig, die RETURN-Taste zu drücken, um eine neue Zeile zu erhalten. Vielmehr erzeugt Edward automatisch beim Umbruch neue Zeilen. Sie brauchen also nur draufloszutippen. Daher stammt auch die Bezeichnung „Fließtext“.

☞ Die Umbruchposition wird durch Angabe des rechten Randes im „Edit Prefs“-Formular angegeben. Der „Word-Wrap“-Modus kann ebenfalls durch dieses Formular ein- und ausgeschaltet werden. Wie Sie dabei vorgehen müssen, haben Sie im Kapitel IX. „Das Edit-Prefs Formular“ gelesen.

Im Gegensatz zu anderen Editoren, die einen Absatz erst durch Wahl eines bestimmten Menüpunktes einmalig umbrechen können, formatiert Edward den Text schon während der Eingabe. Auch beim Löschen innerhalb eines Absatzes werden Worte aus folgenden Zeilen eines Absatzes wieder nachgezogen, wenn in der aktuellen Zeile durch das Löschen Platz für sie gewonnen wurde.

Neben dem automatischen Umbruch des „Word-Wrap“-Modus können Sie einen Text auch nachträglich umformatieren.

Hierzu wählen Sie entweder „Format Paragraph“, um einen Absatz neu umzubrechen oder „Format Text“, um den gesamten Text neu umzubrechen. Beide Menüpunkte befinden sich im „Extras“-Menü.

XI. Makros

1) Makros aufzeichnen & abspielen

2) Makros speichern & abspielen

Eine Besonderheit von Edward, die Ihnen viel Arbeit erleichtern kann, ergibt sich aus der Möglichkeit, Makros aufzeichnen und abspielen zu können.
Fast alle Aktionen, die Sie beim Editieren eines Textes ausführen, können Sie wie mit einem Kassettenrekorder aufnehmen und bei Bedarf wiedergeben lassen. Das bedeutet, daß Sie sich alle gleichförmigen langweiligen Vorgänge, die Sie oft wiederholen müßten, durch Makros abnehmen lassen können.

1) Makros aufzeichnen & abspielen

Wenn Sie ein Makro aufzeichnen wollen, so wählen Sie den Menüpunkt „Aufnehmen" aus dem „Macro"-Menü. Daraufhin werden alle folgenden Aktionen, die Sie vornehmen, aufgezeichnet. Mit „Anhalten" aus dem „Macro"-Menü beenden Sie eine Aufzeichnung. „Abspielen" spielt das Makro dann für Sie ab.

- Laden Sie einen größeren Text ein.

Angenommen Sie wollten nun zu Beginn jeder Zeile die Zeichenfolge „Start>" und am Ende jeder Zeile die Zeichenfolge „<End" einfügen. Normalerweise müßten Sie jedesmal an den Anfang einer Zeile gehen, „Start>" eintippen, dann ans Ende einer Zeile springen, „<End" eingeben und in die nächste Zeile gehen. Wenn Sie die Möglichkeiten der Makros ausnutzen, wird das Ganze für Sie sehr viel einfacher.

- Fahren Sie mit dem Cursor in die erste Zeile Ihres Textes.
- Wählen Sie nun den Menüpunkt „Aufzeichnen" aus dem „Macro"-Menü.

Alle folgenden Aktionen werden nun aufgezeichnet.

- Drücken Sie SHIFT-CursorLeft. Damit springen Sie an den Anfang einer Zeile.
- Tippen Sie nun „Start >" ein.

- Anschliessend bewegen Sie den Cursor mit SHIFT-CursorRight an das Ende der aktuellen Zeile.

- Geben Sie nun „< End“ ein.

- Fahren Sie mit dem Cursor eine Zeile tiefer.

- Wählen Sie „Anhalten“ aus dem „Macro“-Menü. Damit haben Sie die Aufzeichnung beendet. Alle Aktionen zwischen „Aufnehmen“ und „Anhalten“ wurden aufgezeichnet und stehen nun zu Ihrer Verfügung.

Sie können nun jederzeit das gerade aufgezeichnete Makro abspielen.

- Wählen Sie „Abspielen“ aus dem „Macro“-Menü. Sie sehen, wie die folgende Zeile in Sekundenbruchteilen ebenfalls mit „Start >“ und „< End“ eingerahmt wird. Sie können diesen Vorgang nun beliebig oft wiederholen und sich damit viel Tipparbeit ersparen.

Statt obigem zugegebenermaßen etwas sinnlosem Beispiel werden Ihnen in der Praxis sicherlich sehr viel nützlichere Einsatzmöglichkeiten für die Makrofunktionen einfallen.

2) Makros speichern & abspielen

Die mit „Aufnehmen“ aufgezeichneten Makros stehen hier nur kurze Zeit zur Verfügung. Wenn Sie ein neues Makro aufzeichnen oder Edward verlassen, so ist Ihr zuletzt aufgezeichnetes Makro für immer verloren. Damit Sie ein besonders praktisches Makro nicht immer wieder neu aufzeichnen müssen, bietet Ihnen Edward die Möglichkeit ein Makro abzuspeichern und abgespeicherte Makros erneut abzuspielen.

- Falls Sie das Beispiel aus Abschnitt 1) noch nicht durchgespielt haben, so sollten Sie es jetzt tun. Wir benötigen für die folgenden Schritte nämlich ein frisch aufgezeichnetes Makro.

- Wählen Sie nun den Menüpunkt „Speichern als ...“ aus dem „Macro“-Menü. Es erscheint der Dateiauswahlfenster. Speichern Sie das Makro nun unter irgendeinem Namen ab.

- Zeichnen Sie ein anderes Makro auf oder verlassen Sie Edward und starten ihn danach erneut. Dies dient dazu das aktuelle Makro zu „vergessen“. Am einfachsten

ist dies für Sie, wenn Sie irgendwelche neuen Aktionen wie in Abschnitt 1) beschrieben aufnehmen.

- Wählen Sie nun „Open & Play“ aus dem „Macros“-Menü. Mit dem Dateiauswahlfenster geben Sie nun das im vorletzten Schritt abgespeicherte Makro an.

Sie werden sehen, wie Edward automatisch die aktuelle Zeile mit „Start >“ und „< End“ umrahmt.

Somit können Sie auch ein gespeichertes Makro beliebig oft wiederaufrufen.

☛ Ein mit „Öffnen und abspielen“ abgespieltes Makro läßt sich jederzeit mit „Abspielen“ aus dem „Macro“-Menü aufrufen, ohne erneut geladen werden zu müssen. Ein altes von Ihnen aufgezeichnetes Makro wird durch „Öffnen und abspielen“ überspielt.

☞ Im Zusammenhang mit „ARexx“ ergeben sich einige weitere Möglichkeiten. Lesen Sie dazu bitte das Kapitel XXI. „Edward - Kommunikation mit ARexx“.

XII. Fenster

1) Erzeugen von Fenstern

2) Wahl eines bestimmen Fensters

3) Schließen von Fenstern

4) Arbeiten mit mehreren Ansichten eines Textes

Sie wissen jetzt schon, wie man verschiedene Texte bearbeitet. Vielleicht haben Sie sich auch schon gefragt, wie man verschiedene Texte oder mehrere Ausschnitte eines bestimmten Textes gleichzeitig anzeigen kann.
Edward ermöglich dies durch die sogenannten Fenster. Was ein Fenster auf der Workbench oder im CLI bedeutet ist klar. Bei Edward ist ein Fenster ein normales Amiga-Fenster verbunden mit den Informationen, die Edward benötigt, um darin einen bestimmten Text oder Textausschnitt anzuzeigen. Wie das genau funktioniert, erkläre ich Ihnen in den folgenden Abschnitten.

1) Erzeugen von Fenstern

Nach dem Start öffnet Edward sofort ein Fenster, in welchem Sie einen Text bearbeiten können.

- Geben Sie irgendeinen Text ein.

- Wählen Sie aus dem „Fenster"-Menü den Eintrag „Neu". Sie sehen kurz, wie sich ein Fenster öffnet. Dieses Fenster liegt genau über dem ersten Fenster von Edward und hat auch denselben Inhalt. Wenn Sie nur flüchtig zugeschaut haben, werden Sie keinen Unterschied in der Anzeige bemerkt haben.

- In der rechten unteren Ecke befindet sich ein Symbol mit welchem Sie die Größe des Fenster verändern können. Halbieren Sie das Fenster in der Höhe und der Breite. Sie sehen nun, daß darunter Ihr altes Fenster liegt.

- Verschieben Sie das Fenster in die rechte, untere Ecke.

- Wählen Sie den Eintrag „Neu“ aus dem „Text“-Menü.

- Laden Sie einen Text mit „Projekt/Öffnen“, oder geben Sie einen weiteren Text ein.

Für den weiteren Verlauf unserer Übung sollten Sie jetzt noch ein Fenster öffnen, es verkleinern und in die linken unteren Ecke verschieben. Danach laden Sie einen weiteren Text oder Sie geben einfach ein paar Zeilen ein.

2) Wahl eines bestimmten Fensters

Es gibt verschiedene Möglichkeiten, ein bestimmtes Fenster zu aktivieren.

- Am einfachsten fahren Sie mit dem Mauspfeil im entsprechenden Fenster an die Position im Text, die Sie bearbeiten wollen und drücken die linke Maustaste. Das Fenster wird daraufhin aktiviert und der Cursor erscheint unter dem Mauspfeil. Über das Tiefen-Gadget (bzw. bei Kickstart 1.2, 1.3 die beiden Tiefengadgets) können Sie an jedes Fenster herankommen und es dann wie oben beschrieben mit dem Mauspfeil aktivieren.

- Wählen Sie auf diese Weise das Fenster in der rechten unteren Ecke an.

Ebenso wie die Texte werden auch die Fenster in einer Liste verwaltet. Auf einfache Weise können Sie das Fenster an der Listenposition vor oder auch nach dem momentan aktuellen wählen. Hierbei gilt ebenfalls alles zu den Texten gesagte. Das momentan aktuelle Fenster sollte das von Ihnen erzeugte in der linken unteren Ecke sein.

- Wählen Sie aus dem „Fenster“-Menü den Eintrag „Vorheriges“. Damit haben Sie wieder das Fenster in der rechten, unteren Ecke aktiviert.

- Wählen Sie erneut „Vorheriges“. Die beiden kleinen Fenster sind nun plötzlich verschwunden. Einzig das grosse Fenster, das beim Start von Edward automatisch geöffnet wurde, ist nun sichtbar. Das liegt daran, daß bei einer Aktivierung eines Fensters durch „Vorheriges“ oder „Nächstes“ dieses immer automatisch in der Vordergrund geholt wird.

3) Schließen von Fenstern

Sie können ein Fenster auf zwei Arten schließen. Entweder Sie bewegen dem Mauspfeil auf das Schließen-Symbol in der linken, oberen Ecke des Fensters und drücken die linke Maustaste, oder Sie wählen den Eintrag „Schließen" aus dem „Fenster"-Menü, der das momentan aktive Fenster schließt.

- Aktivieren Sie, wie im vorigen Abschnitt beschrieben, das große Fenster.
- Schließen Sie es nun durch den „Schließen"-Menüeintrag. Nun sind nur noch die beiden kleinen Fenster vorhanden.
- Schließen Sie nun das Fenster in der rechten, unteren Ecke mit dem Mauspfeil.

Wenn Sie nun mehrfach „Nächster" oder „Vorheriger" aus dem „Text"-Menü (nicht dem „Fenster"-Menü !) wählen, dann stellen Sie fest, daß die drei verschiedenen Texte alle noch da sind, obwohl Sie zwei Fenster geschlossen haben. Das liegt daran, daß beim Schließen eines Fensters nur die Anzeige eines Textes beendet wird. Der Text wird dadurch aber nicht freigegeben.

☛ Wichtig:
Wenn Sie versuchen, das letzte Fenster von Edward zu schließen, geht Edward davon aus, daß Sie die Arbeit beenden wollen. Sollten Sie noch Texte haben, deren Änderungen nicht abgespeichert wurden, so werden Sie durch ein Dialog-Fenster darauf aufmerksam gemacht. Gegebenenfalls können Sie die Operation mit „Nein" abbrechen. Wählen Sie „Ja", dann gehen alle vorher gemachten Änderungen verloren.

- Lesen Sie hierzu auch Abschnitt 3) aus Kapitel IV „Text".

4) Arbeiten mit mehreren Ansichten eines Textes

Mit Edward können Sie sich mehrere Ansichten desselben Textes anzeigen lassen und damit arbeiten.

- Sie sollten nun nur noch ein Fenster von Edward geöffnet haben. Sollte dies nicht der Fall sein, so sorgen Sie bitte mit den oben beschriebenen Schritten dafür.
- Bewegen Sie nun das Fenster in die linke, obere Ecke des Bildschirms und verändern Sie seine Größe so, daß es die gesamte Breite und etwa ein Drittel der Höhe des Bildschirms ausfüllt.

Damit Sie die in diesem Abschnitt beschriebenen Möglichkeiten richtig kennenlernen können, sollten Sie nun einen Text einladen oder eingeben, der etwa 100 Zeilen oder mehr umfasst.

- Erzeugen Sie zwei weitere Fenster mit denselben oben beschriebenen Ausmassen und ordnen Sie diese untereinander so an, daß es keine Überlappungen zwischen ihnen gibt.

- Aktivieren Sie nun das mittlere Fenster und fahren Sie mit dem Cursor nach unten in die 40zigste Zeile.

- Aktivieren Sie das unterste Fenster und fahren Sie mit dem Cursor in die letzte Zeile des Textes.

Sie können nun in jeder der drei Ansichten des Textes an verschiedenen Positionen weiterarbeiten. Der Vorteil ist, daß Sie in den beiden anderen Fenstern eine Übersicht über ganz andere Textstellen haben, als jene, die Sie gerade bearbeiten. Das kann besonders beim Erstellen von Quellcode für eine bestimmte Programmiersprache von Vorteil sein.

☛ Hinweis:
Alle Änderungen, die Sie an einem Text vornehmen, inklusive der Löschung desselben, werden nur in dem aktuellen Fenster auch angezeigt. Erst wenn Sie eine andere Ansicht des Textes in einem anderen Fenster anwählen, wird auch dort die Anzeige aktualisiert. Lassen Sie sich davon nicht verwirren.

Um Ihnen das Arbeiten mit verschiedenen Ansichten eines Textes zu erleichtern, bietet Edward zusätzliche Funktionen an.

- Schließen Sie alle Fenster bis auf eines.
- Wählen Sie nun „volle Größe“ aus dem „Fenster“-Menü.

Sie sehen, daß das letzte Fenster sich wieder auf die maximale Größe ausdehnt. So erhalten Sie schnell wieder den Überblick, falls ein Fenster zu klein war oder Sie durch Schließen anderer Fenster mehr Platz gewonnen haben.

- Wählen Sie nun „horizontal teilen“ aus dem „Fenster“-Menü. Daraufhin teilt sich das aktuelle Fenster horizontal in zwei gleichgroße Fenster auf. So erhalten Sie schnell die nötigen Fenster für verschiedene Ansichten desselben Textes. Mit „vertikal teilen“ unterteilen Sie genauso ein Fenster vertikal in zwei gleichgroße Fenster.

XIII. Verlassen des Editors

1) Unterbrechen der Arbeit

2) Verlassen des Editors

1) Unterbrechen der Arbeit

Edward bietet Ihnen besonderen Komfort, falls Sie die aktuelle Arbeit an einem oder mehreren Texten unterbrechen und den Editor verlassen wollen. Wenn Sie im „Projek t"-Menü den Eintrag „Projekt & Ende" wählen, so speichert Edward zusätzliche Informationen über alle momentan verwendeten Fenster und Texte ab. Wenn Sie Edward erneut starten, dann öffnet er automatisch Fenster und lädt Texte ein, so daß Sie die Arbeit in derselben Umgebung fortsetzen können, in der Sie sie unterbrochen haben.

☛ Hinweis:
Falls Sie Änderungen an Texten vornahmen, ohne diese zu speichern und daraufhin „Projekt & Ende" gewählt haben, so werden Sie über ein Dialog-Fenster gefragt, ob sie diese Texte „vergessen" wollen. Antworten Sie hier mit „Nein", dann wird die Operation abgebrochen. Wählen Sie „Ja", dann gehen die letzten Änderungen verloren!

- Damit Sie einen kleinen Eindruck über die Möglichkeiten der Arbeitsunterbrechung erhalten, sollten Sie nun Fenster mit verschiedenen Texten erzeugen.

☞ Hierzu können Sie noch einmal in den Kapiteln III. „Eingabe und ..." und XII. „Fenster" nachschauen

- Wählen Sie nun den Menüpunkt „Projekt & Ende" aus dem „Projekt"-Menü. Sie befinden sich nun wieder auf der Workbench oder im CLI, je nachdem, wie Sie Edward gestartet haben.

- Starten Sie Edward erneut. Automatisch öffnen sich die Fenster wieder so, wie Sie vor dem Verlassen angeordnet waren und zeigen die zuletzt bearbeiteten Texte an. Sie können nun einfach mit der Arbeit fortfahren.

2) Verlassen des Editors

Im Gegensatz zu „Projekt & Ende“ verlassen Sie mit „Ende“ aus dem „Projekt“-Menü den Editor, ohne daß Informationen zur momentanen Arbeitsumgebung gespeichert werden.

☛ Hinweis:
Falls Sie Änderungen an Texten vornahmen, ohne diese zu speichern und daraufhin „Ende“ gewählt haben, so werden Sie über ein Dialog-Fenster gefragt, ob sie diese Texte „vergessen“ wollen. Antworten Sie hier mit „Nein“, dann wird die Operation abgebrochen. Wählen Sie „Ja“, gehen die letzten Änderungen verloren!

XIV. Iconify & HotKey

1) Iconify

2) HotKey

1) Iconify

Benutzen Sie Edward auf dem Workbench-Screen, so wird es manchmal sehr lästig sein, wenn Edward mit seinen Fenstern etwas im Hintergrund verdeckt. Vielleicht wollen Sie einfach ein Programm von der Workbench aus starten oder in der Shell den Compiler anwerfen. Damit Sie sich nicht durch alle Fenster Edwards durchklicken müssen, gibt es die Funktion Ikonifizieren
Wählen Sie einfach aus dem „Fenster"-Menü den Eintrag „Ikonifizieren". Schon verschwinden alle Fenster, und es erscheint links oben auf dem Bildschirm ein Fenster, das nur aus einer Titelzeile besteht. In dieser steht „Iconified Edward". Edward ist nun nicht mehr aktiv. Das kleine Fenster könnnen Sie irgendwohin schieben, wo es Sie nicht stören kann.
Wollen Sie Edward wieder aktivieren, so haben Sie zwei Möglichkeiten. Die zweite wird im Abschnitt HotKey beschrieben.
Die erste Möglichkeit besteht darin, das Fenster mit dem Titel „Iconified Edward" zu aktivieren und dann die rechte Maustaste zu drücken. Daraufhin baut Edward wieder alle seine Fenster auf und Sie können weiterarbeiten.

2) HotKey

Um Edward nach einem Iconify wieder aktivieren zu können, oder um einfach das Fenster mit dem zuletzt bearbeiteten Text in den Vordergrund zu bekommen, gibt es den HotKey.
Der HotKey ist eine Taste in Verbindung mit bestimmten Qualifiern (SHIFT, ALT, CTRL, LEFT_AMIGA). Wird diese Taste gedrückt, so wird das Fenster mit dem zuletzt bearbeiteten Text und der Bildschirm auf dem sich dieses Fenster befindet in den Vordergrund gebracht. War Edward im Iconify-Zustand, so wird dieser aufgehoben, und alle Fenster werden wieder geöffnet.

☞ Näheres zur Definition des HotKeys erfahren Sie im Kapitel XX. „Benutzer-definierte Menüs, Tastaturbelegung und HotKey".

XV. Der Kommandozeilen-Interpreter

Die Bedienung Edwards über Menüs und Formulare stellt für den Anfänger sicherlich eine große Erleichterung dar, der Profi ist in vielen Fällen jedoch bereit auf diesen Komfort zu verzichten, wenn er nur besonders schnell die wichtigsten Funktionen benutzen kann.
Dies kann zum einen durch Definition von „Short-Cuts" wichtiger Menüpunkte und durch Belegung von Funktionstasten etc. geschehen. Eine weitere mächtige Möglichkeit ist der Kommandozeilen-Interpreter Edwards. Hier können viele Befehle und Befehlskombinationen Edwards direkt eingegeben und ausgeführt werden.

Die Eingabe einer Kommandozeile leiten Sie durch die „ESC"-Taste ein. Daraufhin erscheint der Cursor in der untersten Zeiledes aktiven Textfensters. Die dort angezeigte Textzeile wird durch eine Leerzeile ersetzt, in der Sie nun Ihre Eingaben machen können. Ein erneutes „ESC" bricht die Eingabe ab. Drücken Sie stattdessen „RETURN" oder „ENTER" werden die von Ihnen eingebenen Kommandos ausgewertet.
Mit „CTRL-O" gelangen Sie ebenfalls in den Eingabemodus des Kommando-Interpreters mit der Besonderheit, daß Ihre letzte Eingabe wieder zur Verfügung steht und geändert werden kann.
„CTRL-G" bewirkt eine erneute Ausführung der zuletzt eingebenen Kommandos.

☞ Weitere Informationen über den syntaktisch korrekten Aufbau einer Kommandofolge erhalten Sie in Kapitel XX. „Benutzerdefinierte Menüs, Tastaturbelegung und HotKey" in den Abschnitten 1) „Kurze Einführung in die EBNF-Darstellung" und 2) „Definition eigener Menüs", insbesondere aus der Definition der „CommandList".

☞ Alle verfügbaren Kommandos Edwards mit Kurzerläuterungen und Parameterbeschreibung sind in Anhang A) enthalten.

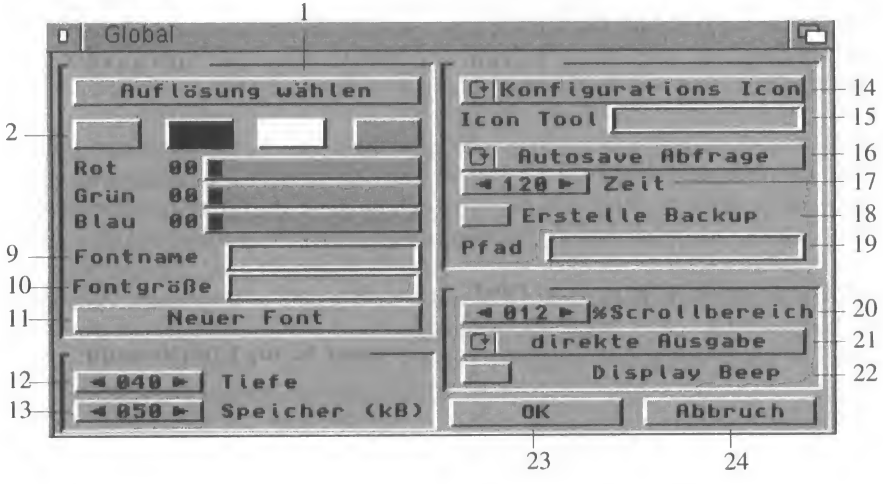
XVI. Globale Einstellungen

1) Das Formular für „globale Einstellungen“

2) Das „Seitenformat“-Formular

1) Das „Globale Einstellungen“-Formular

Mit dem „Globale Einstellungen“-Formular, das sich öffnet, wenn Sie „Einstellungen...“ aus dem „Projekt“-Menü wählen, können Sie verschiedene Einstellungen verändern, die den gesamten Editor betreffen.



1) Bei Klick auf dieses Symbol erscheint ein Dialog-Fenster aus dem Sie die gewünschte Auflösung auswählen können.

Mit den folgenden Symbolen können Sie die vier Farben einstellen, in denen die Fenster und Texte Edwards auf einem eigenen Bildschirm dargestellt werden sollen.

2) - 5) Durch Klicken auf eines dieser Symbole bestimmen Sie die Bildschirmfarbe, die verändert werden soll.

6) - 8) Mit diesen Reglern bestimmen Sie den Rot-, Grün- und Blauanteil der von Ihnen gewählten Farbe. Als kleine Hilfe wird rechts von den Reglern der Wert des jeweiligen Farbanteils zwischen 0 und 15 angezeigt.

9) In dieses Texteingabefeld können Sie den Namen des Zeichensatzes angeben, in welchem die Texte Edwards dargestellt werden sollen.

10) Dieses Texteingabefeld erfaßt die Fontgröße.

☛ Wichtig: Für den beschleunigten Bildaufbau bei der Textdarstellung mußten bei der Entwicklung Edwards gewisse Kompromisse eingegangen werden. Daher ist es leider nicht möglich Proportional-Zeichensätze zu wählen. Sollten Sie einen falschen Zeichensatz gewählt haben, so erfolgt keine Änderung der Textdarstellung.

11) Durch Anklicken dieses Symbols öffnet sich ein Dialog-Fenster, mit dem Sie auf komfortable Weise den gewünschten Zeichensatz wählen können.

12), 13) Mit diesen beiden Zahlenfeldern kontrollieren Sie das Verhalten von Edward im Zusammenhang mit der Undo-Funktion. Mit Undo-Depth geben Sie an, wie viele Operationen, die einen Text verändern, gemerkt werden sollen. Sollte Sie an einem Text eine Anzahl von Änderungen vornehmen, die diesen Wert überschreitet, so werden alte Änderungen „vergessen". Einfache Operationen, wie „Zeichen löschen" oder „Neues Zeichen einfügen" benötigen keinen Zusatzspeicher zum Merken. Das Ausschneiden eines Blockes dagegen kann nur gemerkt werden, wenn genügend Speicherplatz bereitgestellt wurde, um den ausgeschnittenenen Block zusätzlich speichern zu können. Würden Sie nun sehr oft Blöcke ausschneiden und kopieren, so würde sehr viel Speicher in Anspruch genommen. Mit „UndoMem" begrenzen Sie den Speicherplatz, den die Undo-Funktion in Anspruch nimmt, auf einen Maximal-Wert. Wird dieser überschritten, so werden alte Operationen „vergessen", um genügend Speicher wieder freizugeben.

☛ Wichtig: Da die Undo-Funktion für jeden Text die notwendigen Informationen zum Rücknehmen einer Veränderung merken muß, gilt Undo-Mem auch als Wert für jeden Text. Das bedeutet, daß Sie für jeden Text, den Sie bearbeiten, im ungünstigsten Fall die Menge von „UndoMem" zusätzlich an Speicher beanspruchen.

Mit den anschließend beschriebenen Symbolen können Sie das Verhalten Edwards beim Speichern von Dateien angeben.

14) Mit diesem Blätter-Symbol geben Sie an, ob Edward beim Speichern einer Datei ein Pictogramm anlegen („Configuration Icon") oder darauf verzichten soll („No Icon").

☛ Wichtig: Nur wenn Sie die Option „Configuration Icon" gewählt haben, werden beim Sichern einer Datei wichtige Zusatzinformationen ebenfalls mitabgespeichert. Dazu gehören unter anderem die aktuelle Position in Ihrem Text, der „Word-Wrap"-Modus und - sehr wichtig - die Strukturierung des Textes durch ausgezeichnete Zeilen.

15) Wenn Sie ein Pictogramm zu jedem Text abspeichern lassen, so können Sie mit diesem Eingabefeld bestimmen, welches Programm beim Doppelklick auf das Pictogramm des gespeicherten Textes geladen werden soll. Im Normalfall sollte dies „Edward:Edward" sein.

16) Edward kann die von Ihnen bearbeiteten Texte nach einer bestimmten Zeitspanne abspeichern. Mit dem Blätter-Symbol haben Sie die Wahlmöglichkeit zwischen :

- „Auto-Save OFF", das bedeutet das automatische Abspeichern von Texten ist abgeschaltet.
- „Auto-Save", dann speichert Edward nach der in 17) angebenen Zeitspanne alle veränderten Texte automatisch ab.
- „Confirm Auto-Save", dann fragt Edward vor dem automatischen Speichern von Texten um Erlaubnis.

17) Mit diesem Einsteller können Sie die Zeit in Sekunden angeben, die verstreichen darf, bis Edward automatisch die geänderten Texte abspeichert.

18) Hiermit bestimmen Sie, ob beim Speichern einer Datei eine möglicherweise schon vorhandene Datei gleichen Namens in ein anderes Verzeichnis umkopiert und umbenannt werden soll.

19) In diesem Eingabefeld geben Sie ein spezielles Muster an. Dieses besteht aus dem Pfadnamen gefolgt von einer Zeichenkette, die ein „*" enthalten sein kann. Beim Anlegen eines Backups einer Datei wird diese in den angebenen Pfad umkopiert. Daraufhin wird sie nach der oben erwähnten Zeichenkette umbenannt wobei der „*" durch den alten Namen der Datei ersetzt wird.

20) Mit Scrollbereich ist der Bereich innerhalb eines Textfensters gemeint, in dem der Cursor das Scrollen des Textes bewirkt. Sie können den Abstand des Cursors zu den Fenster-Randkomponenten in Prozent angeben.

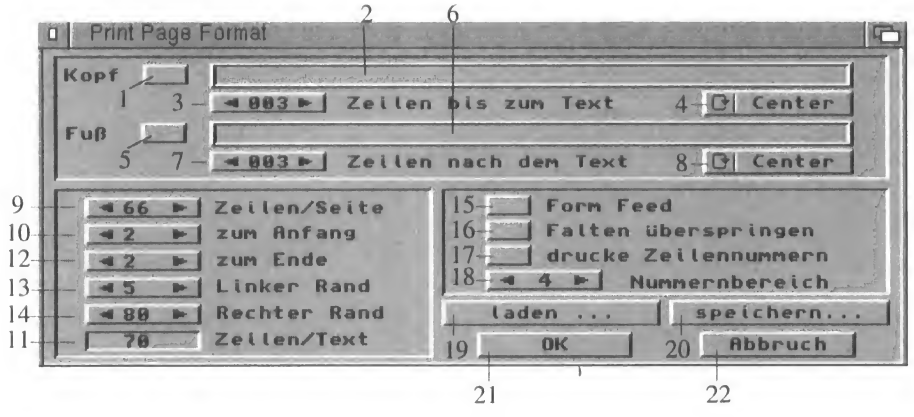
21) Durch das Blätter-Symbol können Sie einstellen, ob die Textausgabe mit Betriebssystem-Funktionen oder direkt gemacht werden soll. Diese Einstellmöglichkeit erhöht die Kompatibilität Edwards zu späteren Betriebssystem-Versionen und zu neuen Grafikkarten mit besonderen Fähigkeiten.

22) Mit diesem Symbol läßt sich der Display-Beep, der durch jeden Fehler erzeugt wird, abschalten.

23),24) Mit diesen beiden Symbolen übernehmen Sie die Einstellungen des Formulars oder Sie brechen den Vorgang ab.

2) Das „Seitenformat"-Formular

Mit dem „Seitenformat"-Menü können Sie bestimmen, wie ein Text auf dem Drucker ausgegeben werden soll.



1), 5) Mit diesen Gadgets geben Sie an ob eine Kopf- bzw. eine Fußzeile ausgedruckt werden soll.

2), 6) Hier wird der Text für die Kopf- bzw. Fußzeile angegeben. Spezielle Kürzel in diesem Text werden beim Druck interpretiert. Möglich sind hier: „%D" = Tag, „%M" = Monat, „%Y" = Jahr, „%h" = Stunde, „%m" = Minuten und „%s" = Sekunden. Durch diese Kürzel kann die aktuelle Zeit beim Ausdruck einer Seite in der Kopf- bzw. Fußzeile eingefügt werden.
Weiterhin kann durch „%p" die Seitennummer eingefügt werden.

3) Gibt die Anzahl Zeilen zwischen der Kopfzeile und dem Text an.

7) Analog zu 3) gibt 7) die Anzahl Zeilen zwischen Text und Fußzeile an.

4), 8) Hiermit läßt sich die Kopf- bzw. Fußzeile zentriert, links- oder rechtsbündig drucken.

9) Mit 9) geben Sie an, wieviele Zeilen inklusive Kopf- und Fußzeile, sowie aller Abstände auf ein Blatt passen.

10) Hiermit defininieren Sie den Abstand in Zeilen vom Blattbeginn zur ersten Zeile, die gedruckt werden soll (Dies kann die Kopfzeile oder die erste Textzeile sein).

11) Dient zur reinen Information. Sie sehen, wieviele Zeilen für die Ausgabe des Textes übrigbleiben.

12) Gibt analog zu 10) den Abstand in Zeilen zum Ende des Blattes an.

13) Der Abstand des linken Randes in Zeichen.
14) Die Breite einer Zeile in Zeichen.
15) Nach dem Drucken aller Seiten wird ein zusätzlicher FormFeed gesendet, wenn Sie diese Option aktivieren.
16) Hiermit geben Sie an, ob beim Druck eingefaltete Zeichen übersprungen werden sollen.
17) Wählen Sie diese Option an, so wird jede gedruckte Zeile durchnumeriert.
18) Die Stellenanzahl einer Zeilennummer kann hier angegeben werden.
19) Laden eines bestimmten Druckformates.
20) Speichern der Einstellungen.
21) Die Einstellungen werden übernommen.
22) Die Einstellungen werden verworfen.

XX. Benutzerdefinierte Menüs, Tastaturbelegung und HotKey

1) Kurze Einführung in die EBNF-Darstellung

2) Definition eigener Menüs

3) Definition der Tastaturbelegung

4) Definition des HotKeys

5) Einladen einer neuen Definition

In der Datei „Edward.DEF" befinden sich die Definitionen für die Menüs sowie die Tastaturbelegung Edwards. Diese sind in weiten Grenzen veränderbar. Der Aufbau der Definitionsdatei lehnt sich dabei an eine sehr primitive Programmiersprache an, besitzt also eine ganz einfache Syntax. Die folgenden Abschnitte werden Ihnen das näher erläutern.

1) Kurze Einführung in die EBNF-Darstellung

Tja, um Ihnen die Syntax der Definitionen ein bißchen näherzubringen muß ich jetzt leider etwas weiter ausholen und die Erweiterte-Backus-Naur-Darstellung einführen. Hinter diesem ominösen Namen, der jedem Informatik-Studenten mehr oder weniger bekannt ist, verbirgt sich eine Darstellungsform für die Syntax von Programmiersprachen. Mächtig kompliziert ist sie deswegen aber nicht.

Eine Definition beginnt immer mit dem Namen des zu Definierenden, daraufhin folgt ein „="-Zeichen. An dieses schließt sich die eigentliche Definition an. Sie gibt im wesentlichen an, welche Zeichenfolgen syntaktisch korrekt sind. Dies geschieht durch direkte Angabe von Zeichenfolgen oder der Angabe einer anderen Definition, in der dann nachgeschaut werden muß. Zeichenfolgen werden immer in Anführungsstrichen wiedergegeben. Jede Definition wird mit einem „." abgeschlossen.

Beispiel 1:

Wort = "Mahlzeit" .

Würde die Syntax unser Sprache nur aus obiger Definition bestehen, so wäre die einzige syntaktisch korrekte Zeichenfolge „Mahlzeit". Zugegebenermaßen etwas dürftig. Mir sind allerdings Zeitgenossen bekannt, deren Sprachschatz nicht signifikant umfangreicher ist.

Beispiel 2:

KlammerAusdruck = "(" Ausdruck ")".
Ausdruck = "Mahlzeit" .

In diesem Beispiel nimmt die Definition von Klammerausdruck bezug auf die Definition von Ausdruck. Der einzige korrekte Klammerausdruck ist "(Mahlzeit)".

Um anzuzeigen, daß mehrere Zeichenfolgen einer Definition genügen, dient folgende Notation : Name = (... | ...). wobei „..." für eine Zeichenfolge oder eine andere Definition steht.

Beispiel 3:

Wunsch = ("Guten Appetit"| KlammerAusdruck).
KlammerAusdruck = "(" Ausdruck ")" .
Ausdruck = "Mahlzeit" .

Ein Wunsch kann laut dieser Definition entweder aus der Zeichenfolge "Guten Appetit" oder der Zeichenfolge "(Mahlzeit)" bestehen.

Die bisher erwähnten Beschreibungsmöglichkeiten für die Syntax einer Sprache würden schon ausreichen, aber um Schreibarbeit zu ersparen, wurden noch einige Abkürzungsmöglichkeiten geschaffen.

Steht auf der rechten Seite einer Definition ein Ausdruck in eckigen Klammern, so bedeutet dies, daß der Inhalt der Klammern stehen kann oder auch nicht.

Beispiel 4:

Satz : "Jetzt geht es " ["endlich "] "zur Sache."

Die beiden korrekten Zeichenfolgen wären : "Jetzt geht es zur Sache." und "Jetzt geht es endlich zur Sache."

Eine letzte Schreibweise bedient sich der geschweiften Klammern. Ein Ausdruck in geschweiften Klammern kann beliebig oft oder auch gar nicht stehen.

Beispiel 5:

Satz : "Hallo " {"Echo "} "."

Korrekte Zeichenfolgen wären : "Hallo .", "Hallo Echo .", "Hallo Echo Echo ." usw..

Mit den obigen Erläuterungen sollten Sie nun den Aufbau der Definitionsdatei verstehen können. Falls Ihnen bisher alles zu theoretisch trocken vorkam, dann sollten Sie sich einfach mal die Definitionsdatei „Edward.DEFS" anschauen. Die Struktur, die die Definitionen haben müssen, sollte Ihnen auch durch einfaches Anschauen schnell klarwerden.

2) Definition eigener Menüs

Die Syntax der Menü-Definitionen ist wie folgt definiert.

```
Menu        = "MENU" MenuTitle { MenuGroup } .
MenuGroup   = "TITLE" QuotedString { MenuItem } .
MenuItem    = ("ITEM" QuotedString CommandList [QuotedString] |
               "EMPTY" ) .

CommandList = ( "(" Command ")" | "(" {CommandList} ")"  ) .
Command     = ( EdwardCommand | CommandList ) .
```

QuotedString ist eine Zeichenfolge zwischen zwei doppelten Anführungszeichen (z.B. "Projekt" für das Projekt-Menü).
EdwardCommand ist ein Kommando des Editors mit Argumenten (falls erforderlich), wie es im Anhang A) „Die Kommandos des Editors" beschrieben wird.

Zur Bedeutung der einzelnen Definitionen :

Menu: Die Menüdefinitionen in der Datei „Edward.DEF“ müssen mit dem Schlüsselwort „MENU“ beginnen. Daraufhin folgen die Definitionen der Einträge (MenuGroup), wobei mindestens ein Eintrag definiert werden muß.

MenuGroup: Eine Menü-Gruppe beginnt mit dem Schlüsselwort „TITLE“, dem sich ein QuotedString anschließt. Die Zeichenfolge des QuotedString (s.o.) gibt den Titel der Menü-Gruppe an, der erscheint, wenn Sie die rechte Maustaste drücken.
Auf die Angabe des QuotedSting können beliebig viele Einträge folgen.

MenuItem: Ein Eintrag in einer Menü-Gruppe (MenuGroup) besteht entweder aus dem Schlüssel-Wort „EMPTY“, dieses dient dazu verschieden Einträge in einer Gruppe voneinander durch einen Balken zu trennen, oder dem Schlüsselwort „ITEM“ und einigen weiteren Angaben. „ITEM“ muß in jedem Fall ein QuotedString folgen. Dieser gibt den Text des Menü-Eintrags an, der erscheint, wenn Sie mit der Maus die entsprechende Menü-Gruppe aufklappen lassen. Daran schließt sich eine CommandList an. Die CommandList ist eine Folge von Kommandos mit Argumenten, die Edward versteht. Diese werden jedesmal ausgeführt, wenn Sie den entsprechenden Eintrag in einer Menü-Gruppe mit der Maus angewählt haben.
Mit einem weiteren wahlweisen QuotedString können Sie ein Zeichen angeben, welches als Short-Cut dienen soll.

3) Definition der Tastaturbelegung

```
Keys          = "KEYS" { ModeList }
ModeQualList  = "("{ Mode | Qualifier | ""}{ KeyDef }")"
Mode          = ( "MODE1" | "MODE2" | "MODE3" )
Qualifier     = ( "ALT" | "SHIFT" | "CTRL" | "LEFT_COMMAND" )
KeyDef        = ( KeyName | QuotedString | "NP" QuotedString )
                CommandList .
```

```
KeyName = (
"F1" | "F2" | "F3" | "F4" | "F5" | "F6" | "F7" | "F8" | "F9" |
"F10" | "ESC" | "BACK" | "DEL" | "HELP" |
"CRSRUP" | "CRSRDOWN" | "CRSRRIGHT" | "CRSRLEFT" |
"TAB" | "RETURN" | "SPACE" | "ENTER" |
"NP_NUML" | "NP_SCRL" | "NP_PRTSC" | "NP_HOME" | "NP_UP" |
"NP_PGUP" | "NP_LEFT" | "NP_RIGHT" | "NP_END" | "NP_DOWN" |
"NP_PGDN" | "NP_INS" | "NP_DEL" )


CommandList = ( "(" Command ")" | "(" {CommandList} ")"  )
Command     = ( EdwardCommand | CommandList )
```

QuotedString ist eine Zeichenfolge zwischen zwei doppelten Anführungszeichen (z.B. „Project“ für das Project-Menü).
EdwardCommand ist ein Kommando des Editors mit Argumenten (falls erforderlich), wie es im Anhang A) „Die Kommandos des Editors“ beschrieben wird.

Wie ein Tastendruck interpretiert werden soll, hängt von verschiedenen Umständen ab. Wichtig ist zunächst einmal, ob die Taste in Kombination mit ALT, CTRL, SHIFT oder LEFT_COMMAND (der linken AMIGA-Taste) betätigt wurde. ALT, CTRL, SHIFT und LEFT_COMMAND werden auch als Qualifier bezeichnet. Für jede beliebige Kombination aus diesen Qualifiern wird der Tastendruck gesondert interpretiert. Das bedeutet, wenn Sie ALT-F1 drücken, geschieht etwas anderes, als wenn Sie ALT-CTRL-F1 drücken. Drücken Sie eine Taste, ohne gleichzeitig eine oder mehre Qualifier-Tasten gedrückt zu haben, so wird auch das gesondert interpretiert. Daraus ergeben sich sechzehn verschiedene Möglichkeiten eine Taste zu belegen (2 * 2 * 2 *2; (mit ALT und ohne ALT) * (mit CTRL und ohne CTRL) * (mit SHIFT und ohne SHIFT) * (mit LEFT_AMIGA und ohne LEFT_AMIGA)).

Das war schon ganz schön kompliziert, aber es wird noch viel schlimmer.
Mit den Befehlen „SetMode1“, „SetMode2“ und „SetMode3“ können Sie einen speziellen Modus setzen. Ist dieser Modus gesetzt, so wird eine daraufhin gedrückte Taste in Kombination mit allen Qualifiern wieder speziell interpretiert. Danach wird der Modus wieder zurückgesetzt.
Wurde eine Taste nicht mit einer CommandList belegt, so wird beim Druck der Taste das ihr entsprechende ASCII-Zeichen in den Text eingefügt. Das bedeutet für Sie eigentlich nur, daß Sie ein „a“ bekommen, wenn Sie die Taste mit dem „a“ drücken. Es sei denn, aus irgendwelchen Gründen hätten Sie diese Taste mit einer CommandList belegt.

Was das alles zu bedeuten hat, muß Ihnen jetzt noch nicht sonnenklar aufgegangen sein, zumindest können Sie jetzt einen Zusammenhang mit der oben angegebenen Syntax erkennen, den ich Ihnen aber gerne noch etwas näher erläutere.

Keys: Jede Definition der Tastaturbelegung muß mit dem Schlüsselwort „KEYS" beginnen. Diesem schließt sich eine beliebige Anzahl von ModeLists an.

ModeQualList: Bei diesem Wort sollte es jetzt bei Ihnen klingeln. In einer ModeQualList geben Sie an, wie eine Taste in Verbindung mit den gesetzten Modes und Qualifiern interpretiert werden soll. Nach der öffnenden runden Klammer „(„ mit der eine ModeQualList beginnt, folgt eine beliebige Kombination aus Mode- und Qualifier-Angaben (z.B.: „(MODE2 ALT SHIFT ..." oder gar nichts.
Es schliessen sich die KeyDefs an und zum Schluß kommt die obligatorische schliessende runde Klammer „)".

KeyDef: Eine KeyDef beginnt mit der Angabe der zu belegenden Taste. Ist dies eine Sondertaste, so geben Sie deren speziellen Namen an (alle Namen sind oben aufgeführt worden). Ansonsten müssen Sie in einem QuotedString das Zeichen angeben, das erscheint, wenn Sie die Taste ohne irgendeinen Qualifier drücken (also ohne SHIFT, ALT, CTRL). Befindet sich die Taste auf dem numerischen Ziffernblock, so geben Sie vor dem QuotedString das Schlüsselwort NP (NP für NumericPad = Zahlenblock) an.
Der Angabe der zu belegenden Taste folgt eine CommandList. Diese gibt die Kommandos an, die Edward nach dem Drücken der entsprechenden Taste ausführen soll.

Damit Sie nun nicht glauben, nur Nobelpreisträger könnten die Tastaturbelegung Edwards ändern, werde ich Ihnen nun am Ende des Abschnittes ein paar Beispiele vorführen.

Beispiel 1 : Zusammenhang zwischen Qualifiern und Taste

```
KEYS
(
    DELETE      (delete)
)
(ALT CTRL
    DELETE (insertchars "Auf 'nem PC wär das das Ende" )
)
```

Mit der obigen Definition wird beim Drücken von DELETE (Taste „DEL“) in Verbindung mit ALT und CTRL der Text „Auf 'nem PC wär das das Ende“ ab der aktuellen Cursor-Position eingefügt. Dieses Beispiel befindet sich schon in der Datei „Edward.DEFS“. Probieren Sie es ruhig mal aus. Drücken Sie dagegen nur DELETE, so wird das Zeichen rechts vom Cursor gelöscht.

Beispiel 2 : Zusammenhang zwischen Modi und Taste

```
KEYS
(CTRL
  "x"   (SetMode1)
)
(MODE1 CTRL
  "c" (insertchars  "EMACS Quit" )
  "s" (insertchars  "EMACS Save" )
)
```

Obige Defnition bewirkt folgendes :
Wenn Sie CTRL-x drücken wird Modus 1 eingestellt. Die nächste Taste wird also gemäß den Angaben für Modus 1 interpretiert. Danach wird automatisch wieder der Normal-Modus eingestellt. Wenn Sie also CTRL-x und gleich anschließend CTRL-C drücken, so wird er Text „EMACS Quit“ rechts vom Cursor eingefügt.
Auch dieses Beispiel ist schon in der Datei „Edward.DEF“ enthalten. Sie können es daher gleich ausprobieren.

Sollten immer noch nicht alle Klarheiten beseitigt sein, dann laden Sie die Datei „Edward.DEF“ einfach ein und schauen sich die Definitionen an, verändern Sie auch ruhig ´mal etwas, aber vergessen Sie nicht, sich eine Sicherheitskopie von „Edward.DEF„ anzulegen.

4) Definition des HotKeys

```
HotKey     = "HOTKEY" Qualifier HotKeyDef
Qualifier  = ( Qualifier | "ALT" | "SHIFT" | "CTRL" |
             "LEFT_COMMAND" | "" ) .
HotKeyDef  = ( KeyName | QuotedString | "NP" QuotedString ) .
```

```
KeyNam = (
"F1" | "F2" | "F3" | "F4" | "F5" | "F6" | "F7" | "F8" | "F9" |
"F10" | "ESC" | "BACK" | "DEL" | "HELP" |
"CRSRUP" | "CRSRDOWN" | "CRSRRIGHT" | "CRSRLEFT" |
"TAB" | "RETURN" | "SPACE" | "ENTER" |
"NP_NUML" | "NP_SCRL" | "NP_PRCSC" | "NP_HOME" | "NP_UP" |
"NP_PGUP" | "NP_LEFT" | "NP_RIGHT" | "NP_END" | "NP_DOWN" |
"NP_PGDN" | "NP_INS" "NP_DEL" ).
```

Wie Sie im Kapitel „XIV. Iconify & HotKey" nachlesen konnten, kann Edward nach Ausführen der Iconify-Funktion wieder durch einen HotKey aktiviert werden. Welche Taste als HotKey dient, wird ebenfalls in der Datei „Edward.DEF" angegeben. Die Syntax hierfür ist sehr einfach. Die Definition des HotKeys beginnt mit dem Schlüsselwort „HOTKEY". Es folgt eine Liste von Qualifiern wie z.B. ALT oder LEFT_AMIGA und schließlich wird die Taste angegeben, die hinfort die Rolle des HotKeys spielen soll.

Beispiel:

```
HOTKEY
  CTRL ALT DELETE
```

Das bedeutet, wenn Sie Edward in den Iconify-Zustand geschickt haben, so können Sie Ihn durch Drücken von CTRL, ALT und der DELETE-Taste wieder aktivieren. Auf einem PC würde diese Tastenkombination übrigens einen Soft-Reset ausführen.

5) Einladen einer neuen Definition

Normalerweise lädt Edward die Definition der Menüs, der Tastatur, etc. gleich beim Start aus der Datei „Edward.DEF". Sie können aber auch später die einzelnen Definitionen ändern, indem Sie eine andere Definitionsdatei einladen. Dies ist durch Wahl von „Open Defs" aus dem „Prefs"-Menü möglich. Es erscheint dann ein Dateiauswahlfenster, mit dem Sie eine andere Definitionsdatei angeben können.

XXI. ARexx - Kommunikation mit Edward

Diese Kapitel kann leider (für Sie) und zum Glück (für mich, den Autor dieses Handbuches) keine Einführung in die Programmiersprache Rexx sein. Dies würde den Rahmen dieses Handbuches bei weitem sprengen.
Außerdem gibt es inzwischen genügend Sekundärliteratur zu diesem Thema. Was soll dann dieses Kapitel, werden Sie sich fragen?
Ganz einfach: Ihnen die Möglichkeiten im Zusammenspiel zwischen Edward und ARexx aufzeigen.
ARexx wurde ja gerade dazu geschaffen, die unterschiedlichsten Programme von außen zu steuern und in ihren Ablauf einzugreifen.
Edward hat auch auf diesem Gebiet eine ganze Menge zu bieten. Das Meiste verbirgt sich jedoch unter der Oberfläche.
Die einzige Funktion, die Sie direkt über Menüs ansprechen können, ist der Menüpunkt „ARexx-Befehl“ aus dem „Macro“-Menü. Wenn Sie diese Funktion wählen, so erscheint ein Dateiauswahlfenster, mit welchem Sie ein ARexx-Programm auswählen können. Dieses wird daraufhin ausgeführt.
Alles Weitere, was sich aus dem Zusammenspiel zwischen ARexx und Edward ergibt, wird erst über die speziellen Kommandos Edwards möglich, mit denen intern alle Funktionen, die Sie z.B. über die Menüs aktivieren, ausgeführt werden. Alle diese Kommandos können auch über ARexx angesprochen werden. Dazu ist es notwendig, in Ihrem ARexx-Programm die folgende Zeile einzufügen :

```
ADDRESS "EDWARD"
```

Alle folgenden Befehle, die ARexx selbst nicht interpretieren kann, werden dann an Edward geschickt. Welche Befehle dies sein können, erfahren Sie im Anhang A) „Die Kommandos des Editors“. Die Befehle in diesem Anhang sind thematisch geordnet. Im Abschnitt XIV. des Anhangs A) finden Sie zudem spezielle Kommandos, die nur im Zusammenspiel mit ARexx sinnvoll sind.

XXII. Zurücknehmen von Text-Änderungen

Mit den Funktionen „Undo" und „Redo" erhalten Sie durch Edward die Retter in der Not, die Sie wahrscheinlich schon oft vermißt hatten. Wie oft hat man doch schon an einem Text etwas geändert, schnell mal was umkopiert oder gelöscht und dann festgestellt, daß das eigentlich nicht gerade die beste Idee war. Was dann? Natürlich könnte man den alten Text wieder einladen und von Neuem beginnen. Wie lästig.
Zum Glück gibt es die Undo-Funktion in Edward. Damit können Sie eine von Ihnen vorgegebene Anzahl aller Operationen, die einen Text verändert haben, wieder rückgängig machen.
Mit „ReDo" ist es Ihnen sogar möglich, die Rücknahme der Änderungen wieder zurückzunehmen (ganz schön komplex).
Natürlich macht Edward das Ganze nicht umsonst.
Um sich alle Änderungen merken zu können, braucht er Speicher. Wieviel, daß können Sie mit den „Global Prefs" ebenso wie die Anzahl der zu merkenden Operationen einstellen.

☞ Siehe hierzu insbesondere Kapitel XVI. Globale Einstellungen 1) Das Formular für globale Einstellungen.

Um einen kleine Eindruck der „UnDo"-Funktion zu erhalten, probieren Sie einfach das folgenden Beispiel aus.

- Geben Sie folgende Zeile ein: „Das Nichts als Negation des Seins mit dem Potential der Realisation der Existenz."
- Wählen Sie nun „UnDo" aus dem „Extras"-Menü.

Daraufhin verschwindet der letzte Buchstabe, den Sie getippt hatten.
Sie können dies wiederholen, bis irgendwann die Meldung „Everything is undone!" in der Titelzeile erscheint.

- Wählen Sie nun „ReDo" aus dem „Extras"-Menü. Daraufhin erscheint der Buchstabe wieder, der durch die vorige Undo-Operation gelöscht wurde.
- Auch „ReDo" können Sie wiederholen, bis wieder die gesamte Zeile dasteht.

Vielleicht laden Sie jetzt mal einen größeren Text ein, schneiden Blöcke aus und kopieren Sie um, löschen einige Zeilen und tippen neue Zeichen ein.
Wenden Sie dann die Undo-Funktion an.

☛ Hinweis:
Auch die Undo-Funktion stößt einmal an ihre Grenzen. Wenn sehr viel Speicher zum Merken von Textveränderungen benötigt wird, dann kann es vorkommen, daß Edward Speicher, der zum Merken älterer Veränderungen benötigt wurde, freigeben muß. Diese Änderungen sind dann nicht mehr rückgängig zu machen.

Weiterhin können nur so viele Änderungen gemerkt werden, wie Sie in den „Global Prefs“ als Undo-Tiefe angegeben haben. Kommen mehr Änderungen zusammen, so werden die ältesten Änderungen vergessen.

Anhang A) „Die Kommandos des Editors“

In diesem Teil des Anhangs erhalten Sie eine thematische Übersicht über alle verfügbaren Kommandos Edwards. Der Hinweis „Referenz“ zu Beginn eines Abschnitts gibt Ihnen das jeweilige Kapitel an, in welchem Sie weitere Informationen zu dem vorgestellten Kommando erhalten können.
Alle Kommandos werden in einem einheitlichen Format dargestellt. Bei der Angabe der Parameter gilt es folgendes zu beachten :
Die Parameter werden von p_0, p_1 bis p_n durchnumeriert. Hinter jedem Parameter wird sein Typ angegeben.

Beispiel:
p_0=STRING
bedeutet, daß der erste Parameter ein String sein soll.

Es gibt folgende Parametertypen :

1) VALUE: Eine ganze Zahl.
2) STRING: Eine durch Anführungszeichen „ begrenzte Zeichenfolge.
3) LIST: Eine durch „(“ und „)“ begrenzte Aufzählung von Elementen. Die Elemente dürfen dabei VALUES, STRINGS und natürlich auch LISTen sein. Im allgemeinen erwarten die Kommandos Edwards jedoch ihre Parameter in einer genau spezifizierten Reihenfolge. Informationen darüber sind dann aber in der Beschreibung des Befehls zu finden. Listen dienen ebenfalls der Rückgabe von Werten, insbesondere im Zusammenhang mit der ARexx-Schnittstelle.

Ein „OPT_“ vor einem Parameter bedeutet, daß er optional ist und auch weggelassen werden darf.

Als Rückgabewert ist außer VALUE, STRING und LIST zusätzlich noch der sogenannte SMALL_STRING möglich. Hierbei handelt es sich um eine Zeichenfolge, die nicht durch Anführungsstriche begrenzt wird. Dies ist allerings nur im Zusammenhang mit der ARexx-Schnittstelle sinnvoll, da die Kommandos Edwards ihre String-Parameter immer als STRING erwarten.

I. Eingabe und Bearbeitung von Texten

Referenz : Kapitel III. Eingabe und Bearbeitung von Texten

Kommando : **BackSpace**
Abkürzung : **Back**
Parameter : -
Funktion :
Löscht das Zeichen links vom Cursor. Steht der Cursor am Anfang einer Zeile, so wird diese mit der vorigen Zeile zusammengefügt.

Kommando : **Delete**
Abkürzung : **Del**
Parameter : -
Funktion :
Löscht das Zeichen, auf dem sich der Cursor befindet. Steht der Cursor am Ende einer Zeile, so wird diese mit der folgenden Zeile zusammengefügt.

Kommando : **KillLine**
Abkürzung : **Kill**
Parameter : -
Funktion :
Entfernt die aktuelle Zeile aus dem Text. Sie wird im Undo-Buffer gespeichert.

Kommando : **UndoLine**
Abkürzung : **Undo**
Parameter : -
Funktion :
Befindet sich der Cursor in der zuletzt geänderten Zeile, so wird diese in ihren alten Zustand vor der Änderung versetzt.
Befindet sich der Cursor in einer anderen Zeile, so wird die zuletzt geänderte Zeile in ihrem Zustand vor der Änderung vor der aktuellen Zeile eingefügt. Dieses Verhalten dient vor allen dazu, eine durch „KillLine“ gelöschte Zeile wieder richtig in den Text einzufügen.

Kommando : **IndentReturn**
Abkürzung : **IRet**
Parameter : -
Funktion :
Die aktuelle Zeile wird an der Cursorposition in zwei Zeilen aufgespalten. Der Teil der aktuellen Zeile, rechts vom Cursor , wird zur neuen aktuellen Zeile und mit Leerzeichen bzw. Tabs genau wie die davorliegende Zeile (die alte aktuelle) eingerückt.

Kommando : **Return**
Abkürzung : **Ret**
Parameter : -
Funktion :
Die aktuelle Zeile wird an der Cursorposition in zwei Zeilen aufgespalten. Der Teil der aktuellen Zeile, rechts vom Cursor, wird zur neuen aktuellen Zeile.

Kommando : **BigStepUp**
Abkürzung : **BSUp**
Parameter : -
Funktion :
Bewegt den Cursor um etwas weniger als die Anzahl der angezeigten Zeilen nach oben.

Kommando : **BigStepDown**
Abkürzung : **BSDown**
Parameter : -
Funktion :
Bewegt den Cursor um etwas weniger als die Anzahl der angezeigten Zeilen nach unten.

Kommando : **MoveUp**
Abkürzung : **Up**
Parameter : p_0=VALUE
Funktion :
Bewegt den Cursor um p_0 Zeilen nach oben.

Kommando : **MoveDown**
Abkürzung : **Down**
Parameter : p_0=VALUE
Funktion :
Bewegt den Cursor um p_0 Zeilen nach unten.

Kommando : **ScreenUp**
Abkürzung : **SUp**
Parameter : p_0=VALUE
Funktion :
Verschiebt den Textausschnitt unter dem Cursor um p_0 Zeilen nach unten.

Kommando : **ScreenDown**
Abkürzung : **SDown**
Parameter : p_0=VALUE
Funktion :
Verschiebt den Textausschnitt unter dem Cursor um p_0 Zeilen nach oben.

Kommando : **MoveLeft**
Abkürzung : **Left**
Parameter : p_0=VALUE
Funktion :
Bewegt den Cursor um p_0 Zeichen nach links. Geht der Cursor über den Anfang einer Zeile hinaus, so wird er an das Ende der davorliegenden Zeile gesetzt.

Kommando : **MoveRight**
Abkürzung : **Right**
Parameter : p_0=VALUE
Funktion :
Bewegt den Cursor um p_0 Zeichen nach rechts. Geht der Cursor über das Ende einer Zeile hinaus, so wird er an den Anfang der folgenden Zeile gesetzt.

Kommando : **GoStringPos**
Abkürzung : **GoSPos**
Parameter : p_0=VALUE
Return : -
Funktion :
Positioniert den Cursor auf das Zeichen an Position p_0 der Zeichenkette der aktuellen Zeile. Die Position wird von 0 an gezählt.

Kommando : **GoDispPos**
Abkürzung : **GoPos**
Parameter : p_0=VALUE
Return : -
Funktion :
Positioniert den Cursor in der aktuellen Zeile auf Position p_0. p_0 wird in diesem Fall von 1 an gezählt und korrespondiert mit der Anzeige in der Titelleiste des Fensters.

Kommando : **Top**
Abkürzung : **T**
Parameter : -
Funktion :
Setzt den Cursor auf den Anfang der ersten Zeile des Textes.

Kommando : **Bot**
Abkürzung : **B**
Parameter : -
Funktion :
Setzt den Cursor auf den Anfang der letzten Zeile des Textes.

II. Verwalten von Texten

Referenz : Kapitel IV. Text

Kommando : **NewText**
Abkürzung : -
Funktion :
Erzeugt einen neuen Text.

Kommando : **FreeText**
Abkürzung : **FTx**
Parameter : -
Funktion :
Gibt den aktuellen Text frei.

Kommando : **PredText**
Abkürzung : **PTx**
Parameter : -
Funktion :
Wählt den Text vor dem aktuellen Text an oder, falls dieser der erste Text in der Liste war, den letzten.

Kommando : **SuccText**
Abkürzung : **STx**
Parameter : -
Funktion :
Wählt den Text nach dem aktuellen Text an oder, falls dieser der letzte Text in der Liste war, den ersten.

Kommando : **SelTextN**
Abkürzung : **STN**
Parameter : p_0=VALUE
Funktion :
Wählt Text Nummer p_0 aus der Liste vorhandener Texte aus.

Kommando : **New**
Abkürzung : **New**
Parameter : -
Funktion :
Löscht alle Zeilen des aktuellen Textes.

Kommando : **Print**
Abkürzung : **Pr**
Parameter : p_0=STRING
Funktion :
Der aktuelle Text wird in die durch p_0 angegebene Datei ausgegeben. Durch Angabe von p_0="ptr" wird auf den durch die Preferences eingestellten Drucker gedruckt.

III. Dateioperationen

Referenz : Kapitel V. Dateioperationen

Kommando : **Save**
Abkürzung : **SA**
Parameter : p_0=OPT_STRING
Funktion :
Speichern des aktuellen Textes.
Wurde p_0 angegeben, so wird der Text unter diesem Namen abgespeichert und als neuer Name des Textes übernommen. Wurde für p_0 ein Leer-String angebenen, so wird der Name des Textes über einen Dateiauswahlfenster vom Benutzer abgefragt.
Wird p_0 nicht mitangegeben, so wird der Text unter seinem bisherigen Namen abgespeichert.

Kommando : **Open**
Abkürzung : **Op**
Parameter : p_0=OPT_STRING
Funktion :
Überladen des aktuellen Textes. Der aktuelle Text wird „vergessen", und der durch p_0 angegebene Text wird geladen. Wird p_0 nicht oder nur ein Leer-String für p_0 angegeben, so wird der Name des Textes über einen Dateiauswahlfenster vom Benutzer abgefragt.

Kommando : **InsertFile**
Abkürzung : **InF**
Parameter : p_0=OPT_STRING
Funktion :
Einfügen eines Textes in den aktuellen Text. Zu p_0 gilt das bereits oben zu „Save" und „Open" Gesagte.

Kommando : **UpdateText**
Abkürzung : **UtX**
Parameter : p_0=Value
Funktion :
Speichert den Text mit der ID p_0, falls er geändert aber noch nicht gespeichert wurde.

Kommando : **UpdateTexts**
Abkürzung : **UtXs**
Parameter : -
Funktion :
Speichert jeden Texte, der geändert aber noch nicht abgespeichert wurde.

IV. Blockoperationen

Referenz : Kapitel VI. Blockoperationen

Kommando : **Mark**
Abkürzung : **Mk**
Parameter : -
Funktion :
Aktiviert den Markierungsmodus für normale Blöcke. Die Cursorposition bei Ausführung von „Mark" wird als Block-Anfang benutzt. Die aktuelle Cursorposition bei Ausführung einer Blockoperation gilt als Block-Ende.
Erneutes Ausführen von „Mark" bewirkt eine Desaktivierung des Markierungsmodus.

Kommando : **MarkVert**
Abkürzung : **MV**
Parameter : -
Funktion :
Aktiviert den Markierungsmodus für Spaltenblöcke. Erneutes Ausführen von „MarkVert" bewirkt eine Desaktivierung des Markierungsmodus.

Kommando : **Cut**
Abkürzung : **X**
Parameter : -
Funktion :
Schneidet dem markierten Block aus dem Text aus und kopiert ihn in das ClipBoard. Beendet den Blockmarkierungsmodus. Der Text vor dem ausgeschnittenen Bereich wird mit dem Text nach dem ausgeschnittenen Bereich zusammengefügt.

Kommando : **Copy**
Abkürzung : **C**
Parameter : -
Funktion :
Kopiert den markierten Block in das ClipBoard. Beendet den Blockmarkierungsmodus.

Kommando : **Paste**
Abkürzung : **V**
Parameter : -
Funktion :
Fügt den im ClipBoard gespeicherten Text an der momentanen Cursorposition ein. Anschließend wird der Cursor auf das letzte eingefügte Zeichen positioniert.

Abkürzung : **PV**
Parameter : -
Funktion :
Fügt die im ClipBoard gespeicherte Textspalte korrekt an der momentanen Cursorposition ein. Anschliessend wird der Cursor auf das letzte eingefügte Zeichen positioniert.

Kommando : **SaveClip**
Abkürzung : **SC**
Parameter : p_0=OPT_STRING
Funktion :
Speichert Text aus dem ClipBoard in einer Datei mit Namen p_0. Wird p_0 nicht oder nur ein Leer-String für p_0 angegeben, so wird der Name der Datei über einen Dateiauswahlfenster vom Benutzer abgefragt.

Kommando : **OpenClip**
Abkürzung : **OC**
Parameter : p_0=OPT_STRING
Funktion :
Lädt Text aus der durch p_0 angebenen Datei und kopiert ihn in das ClipBoard. Wird p_0 nicht oder nur ein Leer-String für p_0 angegeben, so wird der Name der Datei über einen Dateiauswahlfenster vom Benutzer abgefragt.

Kommando : **BlockSL**
Abkürzung : **BSl**
Parameter : -
Funktion :
Rückt alle markierten Zeile eines Blocks um eine Tabulatorweite nach links, falls vor jeder Zeile mindestens ein Tab-Zeichen, bzw. entsprechend viele Leerzeichen sind.

Kommando : **BlockSR**
Abkürzung : **BSr**
Parameter : -
Funktion :
Rückt alle markierten Zeile eines Blocks um eine Tabulatorweite nach rechts.

Kommando : **BlockLower**
Abkürzung : **BLo**
Parameter : -
Funktion :
Wandelt alle markierten Zeichen eines Blocks in Kleinbuchstaben um.

Kommando : **BlockHigher**
Abkürzung : **BHi**
Parameter : -
Funktion :
Wandelt alle markierten Zeichen eines Blocks in Großbuchstaben um.

V. Suchen und Ersetzen

Referenz : Kapitel VII. Suchen und Ersetzen

Kommando : **Find**
Abkürzung : **f**
Parameter : p_0=STRING
Funktion :
Sucht ausgehend von der aktuellen Cursorposition die durch p_0 angebene Zeichenkette im Text. Dabei wird solange in eine Richtung gesucht bis ein Vorkommen der Zeichenkette gefunden werden konnte. Ansonsten wird die Suchrichtung umgekehrt und abgebrochen.

Kommando : **Replace**
Abkürzung : **r**
Parameter : p_0=STRING, p_1=STRING
Funktion :
Sucht ausgehend von der aktuellen Cursorposition die durch p_0 angebene Zeichenkette im Text. Dabei wird solange in eine Richtung gesucht bis ein Vorkommen der Zeichenkette gefunden werden konnte. Ansonsten wird die Suchrichtung umgekehrt und abgebrochen.
Konnte p_0 gefunden werden, so wird sie durch die mit p_1 angegebene Zeichenkette ersetzt.

Kommando : **Again**
Abkürzung : **a**
Parameter : -
Funktion :
Wiederholt die letzte Such- bzw. Ersetzungsoperation.

Kommando : **ReplaceAll**
Abkürzung : **ra**
Parameter : p_0=STRING, p_1=STRING
Funktion :
Ersetzt alle Vorkommen von p_0 im Text durch die Zeichenkette p_1.

Kommando : **RemTab**
Abkürzung : **rt**
Parameter : -
Funktion :
Ersetzt alle Tabs im Text durch entsprechend viele Spaces.

Kommando : **GoMatchStruct**
Abkürzung : **GMS**
Parameter : -
Funktion :
Der Cursor muß sich vor Ausführung von „GoMatchStruct“ auf einer beliebigen öffnenden oder schließenden Klammer befinden. „GoMatchStruct“ bewegt den Cursor dann auf die korrespondieren Klammer.

Kommando : **GoParsedError**
Abkürzung : **GPE**
Parameter : -
Funktion :
Der Cursor muß sich auf der Zeile einer von einem Compiler erzeugten Fehlerdatei befinden. in welcher der Dateiname und die Zeilennummer eines Fehlers stehen. „GoParsedError“ analysiert diese Zeile und prüft, ob die Datei, zu der die Fehlerbeschreibung gehört, sich schon im Speicher befindet. Ist dies nicht der Fall, so wird die entsprechende Datei nachgeladen. Daraufhin wird der Cursor auf die Zeile positioniert, in welcher sich der angebene Fehler befindet.

Kommando : **Jump**
Abkürzung : **J**
Parameter : p_0=VALUE
Funktion :
Positioniert den Cursor auf die durch p_0 angegebene Zeile.

VI. Strukturieren von Texten

Referenz: Kapitel VIII. Strukturieren von Texten

Kommando : **Fold**
Abkürzung : **Fo**
Parameter : -
Funktion :
Faltet einen markierten Block zusammen.

Kommando : **ShowFold**
Abkürzung : **SF**
Parameter : -
Funktion :
Befindet sich der Cursor auf einer geschlossenen Falte, so wird diese hierdurch geöffnet.

Kommando : **RemFold**
Abkürzung : **rf**
Parameter : -
Funktion :
Befindet sich der Cursor auf der ersten Zeile einer Falte, so wird die Falte durch „RemFold" geöffnet, und die Faltenmarkierungen werden entfernt.

Kommando : **SetFoldsOff**
Abkürzung : **SF0**
Parameter : -
Funktion :
Sorgt dafür, daß geschlossene, gefaltete Bereiche mit allen Zeilen genau wie geöffnete, gefaltete Bereiche dargestellt werden.

Kommando : **SetFoldsOn**
Abkürzung : **SF1**
Parameter : -
Funktion :
Macht „SetFoldsOff" rückgängig. Danach werden nur noch explizit mit „ShowFold" geöffnete Falten auch als geöffnet dargestellt.

Kommando : **GoPredFold**
Abkürzung : **GPF**
Parameter : -
Funktion :
Bewegt den Cursor auf den nächstliegenden Faltenanfang oder das nächstliegende Faltenende vor der aktuellen Zeile.

Kommando : **GoSuccFold**
Abkürzung : **GSF**
Parameter : -
Funktion :
Bewegt den Cursor auf den nächstliegenden Faltenanfang oder das nächstliegende Faltenende nach der aktuellen Zeile.

Kommando : **ToggleKey**
Abkürzung : **TK**
Parameter : -
Funktion :
Ändert den Auszeichnungsstatus der aktuellen Zeile. Ist diese ausgezeichnet, so wird die Auszeichnung gelöscht und umgekehrt.

Kommando : **GoPredKey**
Abkürzung : **GPK**
Parameter : -
Funktion :
Bewegt den Cursor auf die nächste vor der aktuellen Zeile liegende Zeile, die ausgezeichnet ist.

Kommando : **GoSuccKey**
Abkürzung : **GSK**
Parameter : -
Funktion :
Bewegt den Cursor auf die nächste der aktuellen Zeile folgende Zeile, die ausgezeichnet ist.

Kommando : **SetBookMark**
Abkürzung : **SBM**
Parameter : p_0=VALUE
Funktion :
Belegt das Lesezeichen mit der Nummer p_0 mit einem Verweis auf die aktuelle Zeile, so daß diese einfach wieder angesprungen werden kann.

Kommando : **GoBookMark**
Abkürzung : **GBM**
Parameter : p_0=VALUE
Funktion :
Sucht, ob das Lesezeichen mit der Nummer p_0 einen Verweis auf eine Zeile enthält und positioniert dann den Cursor auf diese.

VII. Ändern von Optionen, die den aktuellen Text betreffen

Referenz : Kapitel IX. Das Edit-Prefs Formular

Kommando : **SetTab**
Abkürzung : -
Parameter : p_0=VALUE
Funktion :
Setzt die Tabulator-Weite auf p_0. p_0 muß im Bereich von 1 bis 20 liegen.

Kommando : **SetOverOn**
Abkürzung : **SO1**
Parameter : -
Funktion :
Aktiviert den „OverWrite"-Modus.

Kommando : **SetOverOff**
Abkürzung : **SO0**
Parameter : -
Funktion :
Desaktiviert den „OverWrite"-Modus.

Kommando : **SetWrap**
Abkürzung : **SWr**
Parameter : p_0=VALUE
Funktion :
Setzt den rechten Rand für den Umbruch im „Word-Wrap"-Modus auf p_0. p_0 muß größer als 9 sein.

Kommando : **SetWrapOn**
Abkürzung : **sw1**
Parameter : -
Funktion :
Aktiviert den „Word-Wrap"-Modus.

Kommando : **SetWrapOff**
Abkürzung : **sw0**
Parameter : -
Funktion :
Desaktiviert den „Word-Wrap"-Modus.

VIII. „Word-Wrap“

Referenz : Kapitel X. „Word-Wrap“

Kommando : **WrapPara**
Abkürzung : **WrP**
Parameter : -
Funktion :
Formatiert den aktuellen Absatz gemäß den Einstellungen für den „Word-Wrap“ um.

Kommando : **WrapText**
Abkürzung : **WrT**
Parameter : -
Funktion :
Formatiert den aktuellen Text gemäß den Einstellungen für den „Word-Wrap“ um.

XI. Makros

Referenz : Kapitel XI. Makros

Kommando : **RecStart**
Abkürzung : **rst**
Parameter : -
Funktion :
Leitet die Aufzeichnung eines Makros ein. Das zuletzt aufgezeichnete Makro geht dadurch verloren.

Kommando : **RecStop**
Abkürzung : **rsp**
Parameter : -
Funktion :
Beendet die Aufzeichnung eines Makros.

Kommando : **RecPlay**
Abkürzung : **RP**
Parameter : -
Funktion :
Spielt ein aufgezeichnetes oder eingeladenes Makro ab.

Kommando : **SaveRecScript**
Abkürzung : **SRS**
Parameter : p_0=OPT_STRING
Funktion :
Speichert ein aufgezeichnetes Makro in einer Datei mit Namen p_0 ab.
Wird p_0 nicht oder nur ein Leer-String für p_0 angegeben, so wird der Name der Datei über einen Dateiauswahlfenster vom Benutzer abgefragt.

Kommando : **PlayRecScript**
Abkürzung : **PRS**
Parameter : p_0=OPT_STRING
Funktion :
Lädt die durch p_0 angegebene Makro-Datei ein, interpretiert diese und führt sie aus. Wird p_0 nicht oder nur ein Leer-String für p_0 angegeben, so wird der Name der Datei über einen Dateiauswahlfenster vom Benutzer abgefragt.

X. Fenster

Referenz : Kapitel XII. Fenster

Kommando : **NewWin**
Abkürzung : **NW**
Parameter : -
Funktion :
Eröffnet ein neues Fenster.

Kommando : **FreeWin**
Abkürzung : **FW**
Parameter : -
Funktion :
Schließt das aktuelle Fenster. Falls das aktuelle Fenster das letzte Fenster Edwards war, so wird Edward beendet.

Kommando : **PredWin**
Abkürzung : **PW**
Parameter : -
Funktion :
Wählt das Window vor dem aktuellen Window an oder, falls dieses das erste Window in der Liste war, das letzte.

Kommando : **SuccWin**
Abkürzung : **SW**
Parameter : -
Funktion :
Wählt das Window nach dem aktuellen Window an oder, falls dieses das letzte Window in der Liste war, das erste.

Kommando : **SplitHorWin**
Abkürzung : **SHW**
Parameter : -
Funktion :
Teilt das aktuelle Fenster horizontal in zwei gleichgroße Fenster auf.

Kommando : **SplitVerWin**
Abkürzung : **SVW**
Parameter : -
Funktion :
Teilt das aktuelle Fenster vertikal in zwei gleichgroße Fenster auf.

Kommando : **FullSizeWin**
Abkürzung : **FSW**
Parameter : -
Funktion :
Vergrößert das aktuelle Fenster auf maximale Ausmaße.

XI. Verlassen des Editors

Referenz : Kapitel XIII. Verlassen des Editors

Kommando : **Exit**
Abkürzung : **XIt**
Parameter : -
Funktion :
Edward verlassen. Informationen zur Aufnahme der Arbeit in derselben Umgebung, wie vor dem Aufruf von „Exit“ werden gespeichert.

Kommando : **Quit**
Abkürzung : **Q**
Parameter : -
Funktion :
Edward sofort verlassen.

XII. Formulare

Referenz : Kapitel VII. Suchen und Ersetzen
Kapitel IX. Das Edit-Prefs Formular
Kapitel XVI. Globale Einstellungen

Kommando : **FindForm**
Abkürzung : **FFm**
Parameter : -
Funktion :
Öffnet das Such-Formular.

Kommando : **FindRepForm**
Abkürzung : **FRFm**
Parameter : -
Funktion :
Öffnet das Formular zum Suchen und Ersetzen.

Kommando : **GotoForm**
Abkürzung : **GFm**
Parameter : -
Funktion :
Öffnet das Formular, mit welchem eine spezielle Zeile angesprungen werden kann.

Kommando : **GlobalForm**
Abkürzung : **GFm**
Parameter : -
Funktion :
Öffnet das Formular für die globalen Einstellungen.

Kommando : **PageForm**
Abkürzung : **PFm**
Parameter : -
Funktion :
Öffnet das Formular, in welchem den Ausdruck von Texten betreffende Einstellungen vorgenommen werden können.

Kommando : **EditForm**
Abkürzung : **EdFm**
Parameter : -
Funktion :
Öffnet das Formular, mit dem den aktuellen Text betreffende Einstellungen vorgenommen werden können.

Kommando : **InfoForm**
Abkürzung : **InfFm**
Parameter : -
Funktion :
Öffnet das Formular mit Informationen über Edward.

XIII. Der Kommandozeilen-Interpreter

Referenz : Kapitel XV. Der Kommandozeilen-Interpreter

Kommando : **RepeatCmd**
Abkürzung : **RpCmd**
Parameter : -
Funktion :
Führt die zuletzt eingegebene Kommandozeile erneut aus.

Kommando : **EnterCmd**
Abkürzung : **ECmd**
Parameter : -
Funktion :
Wechselt in den Eingabemodus für die Kommandozeile. Ist dieser Eingabemodus aktiv, so bricht erneutes Ausführen von „EnterCmd" den Eingabemodus ab, ohne daß die eingebene Zeile ausgeführt wird.

Kommando : **ModifyOldCmd**
Abkürzung : **MCmd**
Parameter : -
Funktion :
Wechselt in den Eingabemodus für die Kommandozeile, wobei die zuletzt eingegebene Kommandozeile erneut zur Bearbeitung bereitgestellt wird. Ist dieser Eingabemodus aktiv, so bricht erneutes Ausführen von „ModifyOldCmd" den Eingabemodus ab, ohne daß die eingebene Zeile ausgeführt wird.

Kommando : **Do**
Abkürzung : **Do**
Parameter : p_0=VALUE, p_1=CMDLIST
Funktion :
Führt die durch p_1 angegebene Kommandofolge p_0-mal aus.

XVI. Ändern von globalen Einstellungen

Referenz: Kapitel XVI. Globale Einstellungen
Kapitel XX. Benutzerdefinierte Menüs, Tastaturbelegung und HotKey

Kommando : **Display**
Abkürzung : **Dsp**
Parameter : p_0=VALUE
Funktion :
Durch „Display" wird der Screen gewählt, auf welchem die Fenster von Edward erscheinen sollen. Dabei sind die Werte von p_0 folgendermaßen zugeordnet :

0 = Workbench-Screen
1 = 4 farbiger Screen, Hires
2 = 4 farbiger Screen, Hires-Interlaced

Kommando : **Font**
Abkürzung : **Fnt**
Parameter : p_0=STRING, p_1=VALUE
Funktion :
p_0 muß den Namen eines Zeichensatzes angeben (Bsp. „topaz.font"). p_1 gibt die gewünschte Größe des Zeichensatzes an. Bei der Auswahl der Zeichensätze gelten gewisse Einschränkungen. So sind z.B. keine proportionalen Zeichensätze möglich. Sollte Edward aus irgendeinem Grund einen gewünschten Zeichensatz für ungeeignet halten, so wird der zuvor eingestellte Zeichensatz weiterhin verwendet.

Kommando : **AutoMode**
Abkürzung : **Auto**
Parameter : p_0=VALUE, p_1=OPT_VALUE
Funktion :
p_0 wählt das Verfahren für den Auto-Save, p_1 gibt optional die Zeitspanne in Sekunden zwischen zwei Sicherungen an. p_1 muß einen Wert größer oder gleich 30 haben.

Die Werte von p_0 entsprechen folgenden Modi :
0 = Auto-Save OFF
1 = Auto-Save (automatisch speichern, ohne Bestätigung)
2 = Confirm Auto-Save (automatisch speichern, aber erst nach Bestätigung)

Kommando : **OpenDefs**
Abkürzung : **OD**
Parameter : p_0=OPT_STRING
Funktion :
Die Definition der Menüs, Tastaturbelegung, des HotKeys, etc. wird aus der durch den String p_0 spezifizierten Datei geladen. Wird kein String angegeben, so öffnet sich ein Dateiauswahlfenster, mit dem eine Datei gewählt werden kann.

XV. ARexx - Kommunikation mit Edward

Referenz : Kapitel XXI. ARexx - Kommunikation mit Edward

Kommando : **ExecRexx**
Abkürzung : **RX**
Parameter : p_0=OPT_STRING
Funktion :
Führt die durch p_0 angebene Datei als Rexx-Programm aus. Wird p_0 nicht oder nur ein Leer-String für p_0 angegeben, so wird der Name der Datei über einen Dateiauswahlfenster vom Benutzer abgefragt.

Kommando : **ExecRexxString**
Abkürzung : **RXS**
Parameter : p_0=STRING
Funktion :
Führt den übergebenen String p_0 als kleines Rexx-Programm aus.

1) Abfrage von Statuswerten Edwards :

Kommando : **GetWord**
Abkürzung : **GWd**
Parameter : -
Return : SMALL_STRING
Funktion :
Liefert das Wort, auf welchem sich der Cursor befindet zurück.

Kommando : **GetLineString**
Abkürzung : **LineString**
Parameter : -
Return : STRING
Funktion :
Gibt den Inhalt der aktuellen Zeile zurück.

Kommando : **GetTextPos**
Abkürzung : **TextPos**
Parameter : -
Return : LIST „(LineNum, DispPos, DispLineNum, EdStringPos, Fold, FoldStart, FoldEnd, Key, EndLF, EndEOS)“
Funktion :
Liefert Liste, die die Position im Dokument und Status-Informationen enthält.

Aufbau der Liste :

- Zeilennummer; ergibt sich durch Abzählen aller Zeilen vom Textanfang an.
- Spaltenposition
- Zeilennummer; ergibt sich durch Abzählen nur der sichtbaren Zeilen (d.h.: die eingefalteten Zeilen werden nicht mitgezählt).
- Position im String der aktuellen Zeile, die mit der Spaltenposition korrespondiert. Enthält ein String TAB-Zeichen, so können sich die Spaltenposition und die String-Position unterscheiden, da ein Tab für die Spaltenposition normalerweise wie mehrere Leerzeichen gezählt wird.
- Fold. Flag; gibt an, ob die Zeile eingefaltet ist.
- FoldStart. Flag; gibt an, ob die Zeile der Beginn einer Falte ist.
- FoldEnd. Flag; gibt an, ob die Zeile das Ende einer Falte ist.
- Key. Flag; gibt an, ob die Zeile ausgezeichnet wurde.
- EndLF. Flag; gibt an, ob die Zeile mit einem LineFeed (ASCII 10) endet.
- EndEOS. Flag; gibt an, ob die Zeile mit einem EOS (ASCII 0) endet.

Kommando : **GetTextList**
Abkürzung : **TextList**
Parameter : -
Return : LIST „([TextName ID]*)“
Funktion :
Liefert eine Liste aller geladen Texte und ihrer IDs.

Kommando : **GetWindowList**
Abkürzung : **WindowList**
Parameter : -
Return : LIST „([WindowID TextID Left Top Width Height]*)“
Funktion :
Liefert eine Liste mit IDs aller geöffneten Windows, sowie dazugehörigen Informationen (ID des dargestellten Textes, Position und Ausmaße des Fensters).

Kommando : **GetBlockInfo**
Abkürzung : **BlockInfo**
Parameter : -
Return : LIST „(Block, MarkVert, LineNum, DispPos, StringPos)“
Funktion :
Gibt Informationen über einen möglicherweise markierten Block zurück. Ein markierter Block wird immer durch die zu Beginn markierte Zeile und die aktuelle Zeile definiert.

Die Liste enthält folgende Werte:

- Block; Flag. Gibt an, ob ein Block markiert ist. Ist dieses Flag 0, so gelten alle folgenden Informationen nicht.
- MarkVert; Flag. Gibt an, daß der Benutzer einen Spaltenblock markieren will.
- LineNum. Zeilennummer der Zeile, an der die Markierung beginnt (diese kann über-, aber auch unterhalb der aktuellen Zeile liegen).
- DispPos. Spaltenposition des Markierungsbeginns.
- StringPos. Zu DispPos korrespondierende Position im String der Zeile, an der die Markierung beginnt.

Kommando : **GetTextInfo**
Abkürzung : **TextInfo**
Parameter : -
Return : LIST „(TotalLines, DispLines, FoldedLines, UseFold, DoWrap, Over, LockCnt, Changes.)“
Funktion :
Liefert allgemeine Informationen über den aktuellen Text.

Aufbau der Liste :

- TotalLines; Anzahl aller Zeilen des Textes
- DispLines; Anzahl der momentan darstellbaren Zeilen.
- FoldedLines; Anzahl der eingefalteten Zeilen.
- UseFold; Flag. Gibt an, ob Zeilen innerhalb von Falten dargestellt werden sollen (0) oder nicht(-1).
- DoWrap; Flag. Gibt an, ob der WordWrap-Modus eingeschaltet ist.
- Over; Flag. Gibt an, ob der OverWrite-Modus eingeschaltet ist.
- LockCnt; Zähler, wie oft der Text gelocked wurde. Ist er ungleich 0, so kann der Text nicht modifiziert werden. Der LockCnt wird hochgezählt, wenn der Text gedruckt werden soll, oder TextReads auf den Text ausgeführt werden.

Kommando : **GetScreenInfo**
Abkürzung : **ScreenInfo**
Parameter : -
Return : LIST „(Width Height Depth ScreenAdr)“
Funktion :
„ScreenInfo“ gibt folgende Werte in einer Liste zurück :
Breite, Höhe, Tiefe und Adresse des Screens, auf welchem Edward seine Fenster darstellt.

2) Laden von Texten aus Edward heraus

Mit den folgenden Befehlen ist ein Zugriff auf im Speicher befindliche Texte Edwards auf einfache und schnelle Weise möglich.
Zu beachten ist hier vor allem, daß ein von einer externen Applikation mit InitTextRead geöffneter Text bis zum Abschluss der TextRead- Operationen durch EndTextRead nicht verändert werden kann; der Text ist gesperrt („gelocked“).
Solange nicht alle InitTextReads durch entsprechende EndTextReads abgeschlossen wurden, kann zudem Edward nach einem Beenden nicht aus dem Speicher entfernt werden.
Es ist selbstverständlich möglich, mehrere InitTextReads auf denselben Text auszuführen.

Kommando : **InitTextRead**
Abkürzung : **ITR**
Parameter : p_0=STRING
Return : Handle=VALUE
Funktion :
Versucht Text mit Namen p_0 zu finden. Ist dieser vorhanden, so wird er gelocked und ein Handle für weitere Operationen erzeugt. Die erste Zeile des Textes wird zur aktuellen Zeile für kommende TextRead-Operationen. Das Handle wird zurückgegeben.

Kommando : **TextRead**
Abkürzung : **TR**
Parameter : Handle=VALUE, Buffer=VALUE, BuffSize=VALUE
Return : RealRead=VALUE
Funktion :
Kopiert von der für diesen TextRead aktuellen Zeile den Inhalt dieser und der folgenden Zeilen in den Buffer. Dabei werden Zeilen immer komplett kopiert. Sollte eine Zeile nicht mehr ganz in den Buffer passen (daher muß BuffSize angegeben werden), so wird die Kopieraktion beendet.
Daraufhin wird die Anzahl der kopierten Bytes zurückgegeben (RealRead). Das Lesen ist beendet, wenn RealRead gleich 0 ist.

Kommando : **EndTextRead**
Abkürzung : **ETR**
Parameter : Handle=VALUE
Return : -
Funktion :
Der durch ein „InitTextRead" gelockte Text wird wieder unlocked. Das Handle wird freigegeben.

XVI. UnDo/ReDo - Zurücknehmen von Text-Änderungen

Referenz: Kapitel XXII. Zurücknehmen von Text-Änderungen

Kommando : **UnDo**
Abkürzung : **UnDo**
Parameter : -
Funktion :
Nimmt die letzte Änderung am Text zurück.

Kommando : **ReDo**
Abkürzung : **ReDo**
Parameter : -
Funktion :
Nimmt die Rücknahme der Rückname der letzten Textänderung zurück.
(Gut, das ist nicht gerade leichverständlich, aber Sie können ja noch einmal in das Referenzkapitel blättern. Noch besser ist, Sie probieren es einfach aus.)

XVII. Verschiedenes

Referenz : Kapitel XIV. Iconify & HotKey

Kommando : **ExecCli**
Abkürzung : **Exe**
Parameter : p_0=STRING
Funktion :
Führt das Programm mit Namen p_0 aus.

Kommando : **InsertChars**
Abkürzung : **IC**
Parameter : p_0=STRING
Funktion :
Fügt die durch p_0 spezifizierten Zeichen an der aktuellen Cursorposition ein. Dieses Kommando wurde für Makros und die ARexx-Kommunikation eingeführt.

Kommando : **SetMode1, SetMode2, SetMode3**
Abkürzung : **SM1, SM2, SM3**
Parameter : -
Funktion :
Sorgt dafür, daß die nächste Tastatureingabe gemäß Modus 1 interpretiert wird.

Kommando : **Iconify**
Abkürzung : **Ify**
Parameter : -
Funktion :
Schließt alle Fenster Edwards. Ein neues Fenster, das nur aus einer Titelzeile besteht erscheint anschließend. Wird dieses Fenster aktiviert und die rechte Maustaste gedrückt, so werden alle Fenster Edwards wieder an ihren ursprünglichen Positionen geöffnet. Edward kann ebenfalls durch den HotKey oder das Senden einer Nachricht über ARexx „geweckt“ werden.

Anhang B) Fehlermeldungen.

1) Informationen

2) Allgemeine Fehlermeldungen

3) Fehlermeldungen mit Abfragen

4) Meldungen mit Abfragen

5) Fehler bei Betriebssystemoperationen

6) Eingabefehler

7) Fehler in den Definitionsdateien

8) Fehler beim Analysieren von Kommandos

9) Fehlermeldungen beim Auswerten von Kommandos

Beim Arbiten mit Edward werden Sie bei wichtigen Problemen, die sich ergeben können, augenblicklich informiert und erhalten in den meisten Fällen die Möglichkeit, auf diese Informationen zu reagieren. In der Regel handelt es sich hier um Sicherheitsabfragen, um zu verhindern, daß Sie z.B. aus Versehen einen noch nicht abgespeicherten Text löschen.

1) Informationen

Meldung : **„Markierung gesetzt“**

Ursache : Sie haben die aktuelle Zeile mit einem Lesezeichen belegt, um sie demnächst leicht wieder anspringen zu können.

Referenz : Kapitel VIII. 4) Lesezeichen

2) Allgemeine Fehlermeldungen

Meldung : **„Text ist geschützt“**
Ursache : Sie wollten den Text verändern, dieser ist aber gesperrt. Ein Text wird gesperrt, wenn er gedruckt werden soll, oder der Befehl „InitTextRead“ auf ihn ausgeführt wurde.

Abhilfe : Warten, bis der Text gedruckt wurde.
Vielleicht wird auch gerade ein ARexx-Makro auf diesen Text ausgeführt, dann heißt es ebenfalls abwarten.

Referenz : Kapitel IV. 5) Drucken eines Textes
Anhang A) XV.) ARexx - Kommunikation mit Edward

Meldung : **„Kein Pictogramm vorhanden“**

Ursache : Wenn Sie im Formular für die globalen Einstellungen "Configuration Icon" gewählt haben, versucht Edward zu jedem Text, den Sie erzeugen, eine ".info"-Datei zu schreiben. Diese Datei enthält unter anderem Informationen, damit die Workbench ein Pictogramm zu Ihrem Text darstellen kann. Existierte bisher keine ".info"-Datei, so erzeugt Edward eine Standard-Datei. Dazu benötigt er im Verzeichnis "Edward:" ie Datei "EdwardDefault.info". Diese haben Sie anscheinend gelöscht.

Abhilfe : Kopieren Sie von der mitgelieferten Diskette die Datei "EdwardDefault.info" wieder in Ihr "Edward:"-Verzeichnis, sollte Ihnen das zu kompliziert erscheinen müssen Sie Edward komplett neu installieren (Achtung: Dabei können möglicherweise verschiedene von Ihnen vorgenommene Voreinstellungen für Edward verloren gehen).

Referenz : Kapitel XVI. "Globale Einstellungen"

Meldung : **„Abbruch! Konnte Backup nicht erzeugen!“**

Ursache : Sie wollten einen Text speichern. Beim Anlegen der Backup-Datei Ihres Textes trat ein Fehler auf.
Möglicherweise haben Sie einen ungültigenPfad für die Backup-Datei angegeben.
Vielleicht ist auch nur das Laufwerk oder die Platte voll, auf welcher Sie die Backup-Datei anlegen lassen wollten.

Abhilfe : Überprüfen Sie den Pfad-Namen, den Sie für die Backup-Datei im "Global Prefs"-Formular angegeben haben.
Versichern Sie sich, daß auf dem durch den Pfad angegebenen Medium genügend Speicherplatz vorhanden ist.

Referenz : Kapitel XVI. "Globale Einstellungen"

Meldung : **„Markierung nicht gesetzt!“**

Ursache : Sie haben versucht ein Lesezeichen anzuspringen, das noch nicht definiert wurde.

Abhilfe : Vielleicht haben Sie nur die falsche Taste gedrückt, dann sollten Sie es einfach nochmal versuchen.

Referenz : Kapitel VIII. "Strukturieren von Texten - Lesezeichen" Abschnitt 4)

Meldung : **„Kein Block definiert!“**

Ursache : Sie haben versucht eine Block-Operation auszuführen (Cut, Copy, Paste, etc.) , zuvor jedoch keinen Block definiert.

Abhilfe : Einfach den gewünschten Text-Bereich markieren und nochmal versuchen.

Referenz : Kapitel VI. "Blockoperationen"

Meldung : **„Abbruch! Funktion würde Einfaltung zerstören!“**

Ursache : Im allgemeinen wird diese Meldung ausgegeben, wenn Sie einen Block ausschneiden wollten, der nur teilweise eine Text-Falte überdeckt. Bei einer solchen Operation würde nur der Anfang bzw. das Ende der Textfalte ausgeschnitten, was die Falte zerstören würde. Deshalb unterbindet Edward solche Aktionen.

Abhilfe : Sie können eine Textfalte nur komplett ausschneiden. Sollten Sie tatsächlich nur einen Teil der Textfalte ausschneiden wollen, so müssen Sie die Faltung erst rückgängig machen, bevor Sie diesen Teil ausschneiden können.

Referenz : Kapitel VI. "Blockoperationen"

Meldung : **„Zeichenkette zu lang!“**

Ursache : Aus internen Gründen kann Edward maximal 512 Zeichen in einer Zeile verwalten. Beim Einfügen von Zeichen durch eine "Paste"-Operation könnte diese Grenze möglicherweise überschritten werden, was fatale Fehler hervorrufen würde. Daher stoppt Edward hier das Einfügen und gibt obige Meldung aus.

Abhilfe : Die "Paste"-Operation in einer leeren Zeile ausführen.

Referenz : Kapitel VI. "Blockoperationen"

Meldung : **„Suchmuster nicht gefunden!“**

Ursache : Beim Suchen oder Ersetzen konnte das von Ihnen angegebene Suchmuster nirgendwo im Text entdeckt werden.

Abhilfe : Suchvorgang wiederholen. Edward durchsucht den Text immer von der aktuellen Position aus entweder zum Anfang oder zum Ende hin nach dem Suchmuster. Konnte dieses nicht gefunden werden, so kehrt er die Suchrichtung um. Daher kann unter Umständen nach dem Fehlschlagen einer Suchaktion durch eine Wiederholung des Suchvorgangs doch noch etwas gefunden werden. Schlägt dieser Suchvorgang ebenfalls fehl. Dann ist beim besten Willen nichts zu finden.
Vielleicht verbirgt sich eine Zeile, deren Inhalt dem Suchmuster entsprechen würde, in einer Textfalte, dann konnte Edward hier ebenfalls nichts finden. Wählen Sie "Show All Folds" aus dem "Folds"-Menü und wiederholen Sie den Suchvorgang.

Referenz : Kapitel VII. "Suchen und Ersetzen"

Meldung : **„Cursor steht auf keinem Strukturelement“**

Ursache : Beim Aufruf von "Match Bracket" aus dem "Search"-Menü stand der Cursor auf keiner Klammer.

Abhilfe : Positionieren Sie den Cursor auf eine Klammer. Wählen Sie dann erneut "Match Bracket".

Referenz : Kapitel VII. "Suchen und Ersetzen"

Meldung : **„Strukturen stimmen nicht überein!“**

Ursache : Nach dem Aufruf von "Match Bracket" aus dem "Search"-Menü konnte Edward keine korrespondierende öffnende bzw. schließende Klammer finden.

Abhilfe : Überprüfen Sie Ihren Text genau. Sie haben sicher irgendwo eine Klammer vergessen.

Referenz : Kapitel VII. "Suchen und Ersetzen"

Meldung : **„Alles ist rückgängig gemacht worden!“**

Ursache : Sie haben "Undo" aus dem "Extras"-Menü so oft ausgewählt, daß es nichts mehr rückgängig zu machen gibt.

Abhilfe : Wählen Sie doch mal eine andere Funktion aus und bearbeiten Sie Ihren Text wieder.
Na ja, ´mal ganz ernsthaft. Die obige Meldung ist eigentlich eher als Hinweis denn als Fehler zu werten. Nehmen Sie sie einfach zur Kenntnis.

Referenz : Kapitel XXII. "Zurücknehmen von Text-Änderungen"

Meldung : **„Alles ist wiederhergestellt worden!“**

Ursache : Sie haben alle Rücknahmen von Text-Änderungen wieder zurückgenommen, das heißt, Ihr Text sieht wieder genauso aus, wie vor dem ersten Aufruf von "Undo".

Abhilfe : Siehe oben.

Referenz : Kapitel XXII. "Zurücknehmen von Text-Änderungen"

Meldung : **„Menü ist zu groß für den Screen! Nutze Default Einstellungen!“**

Ursache : Sie haben alle Rücknahmen von Text-Änderungen wieder zurückgenommen, das heißt, Ihr Text sieht wieder genauso aus, wie vor dem ersten Aufruf von "Undo".

Abhilfe : Siehe oben.

3) Fehlermeldungen mit Abfragen

Meldung : **„[1][Vorsicht: Kein Speicher mehr!|Texte entfernen|oder Programm verlassen.][OK]“**

Ursache : Edward hat leider nicht genügend Speicher bekommen.

Abhilfe : Sicherheitshalber sollten Sie alle Texte erst einmal abspeichern. Wenn Sie dann einen oder mehrere der momentan nicht mehr benötigten Texte löschen, dann haben Sie für 's erste genügend Speicher zum Weiterarbeiten. Sollte diese Meldung öfter auftreten, dann ist der Kauf einer Speichererweiterung mehr als nur ratsam.

4) Meldungen mit Abfragen

Meldung : **„Geänderten Text verwerfen ?**
[Ja] [Nein]“

Ursache : Sie wollten eine Operation vornehmen, deren Ausführung dazu führen würde, daß der momentan aktuelle Text verloren geht (z.B.: „Open“ aus dem „Projekt“-Menü oder „Neu“ bzw. „Frei“ aus dem „Text“-Menü). Falls dieser Text von Ihnen verändert, aber nicht abgespeichert wurde, so erfolgt diese Meldung. Wenn Sie „Nein“ wählen, so wird die Operation abgebrochen und Sie können den Text danach abspeichern, bei „Ja“ geht dieser Text unwiederbringlich verloren.

Abhilfe : Vor einer der oben erwähnten Operationen den Text abspeichern. Dies ist allerdings nicht zwingend notwendig. Die obige Meldung dient ja gerade dazu, Schlimmeres zu verhindern, wenn Sie das Abspeichern einmal vergessen sollten.

Referenz : Kapitel IV. „Text“, Kapitel V. „Dateioperationen“

Meldung : **„Geänderten Text verwerfen ?**
[Ja] [Nein]“

Ursache : Sie wollten Edward beenden, haben aber noch nicht alle geänderten Texte abgespeichert.

Abhilfe : Wählen Sie „Ja“, dann wird Edward beendet und alle von Ihnen bearbeiteten Texte werden nicht abgespeichert und gehen verloren. Da dies normalerweise nicht gewünscht ist sollten Sie „Nein“ wählen und dann die entsprechenden Texte abspeichern.

Referenz : Kapitel V. „Dateioperationen“

Meldung : **„Altes Pictogramm ersetzen ?**
[Ja] [Nein]“

Ursache : Sie haben eine Datei bearbeitet, zu der ein Icon existiert, das nicht von Edward erzeugt wurde. Dies ist z.B. möglich, wenn Sie eine Script-Datei eines anderen Programmes bearbeiten. Da viele Programme in der „.info“-Datei die das Icon enthält wichtige Zusatzinformationen speichern (diese können Sie sich durch einfaches Anklicken des Icons und Wahl des Menüpunktes „Info“ aus einem Menü der Workbench anzeigen lassen) sollten Sie bei dieser Meldung vorsichtshalber „Nein“ wählen. Nur wenn Sie ganz sicher sind, daß die „.info“-Datei keine wichtigen Informationen enthält wählen Sie „Ja“, dann überschreibt Edward diese Datei mit seiner eigenen „.info“-Datei, die Informationen über das gerade bearbeitete Dokument enthält.

Abhilfe : Vergewissern Sie sich vor der Bearbeitung einer Datei, ob eine „.info“-Datei zu Ihr existiert und ob diese von Edward oder einem fremden Programm erzeugt wurde. Ist letzteres der Fall, so sollten Sie mit dem „Info“-Menü der Workbench erkunden, ob diese Datei irgendwelche wichtigen Zusatzinformationen enthält.

Referenz : Kapitel V. „Dateioperationen“,
Kapitel XVI. „Globale Einstellungen“ Abschnitt 1) „Das ‘Global Prefs’-Formular“

Meldung : **„Datei speichern : 'Name'**
[Ja|Nein]“

Ursache : Sie haben im „Global Prefs“-Formular die Option „Confirm Autosave“ eingestellt. Dies bedeutet, daß Edward nach dem von Ihnen eingestellten Zeitintervall alle Texte überprüft und dann versucht, die geänderten Texte abzuspeichern. Vorher holt er sich jedoch von Ihnen über obige Meldung die Erlaubnis.

Abhilfe : Wenn Sie diese Abfrage stört, dann wählen Sie statt „Confirm Auto-Save“ die Option „Auto-Save“, dann werden geänderte Texte automatisch ohne Rückfrage gespeichert.

Referenz : Kapitel XVI. „Globale Einstellungen“ Abschnitt 1) „Das 'Global Prefs'-Formular“

Meldung : **„Datei existiert bereits**
Überschreiben?
[Ja] [Nein]“

Ursache : Vermutlich haben Sie nach Wahl des Menüpunktes „Speichern als...“ aus dem „Projekt“-Menü im Datei-Auswahlfenster den Namen einer Datei gewählt, die schon existiert. Diese würde von momentan aktuellen Text überschrieben, wenn Sie „Ja“ wählen und ginge auf ewig verloren. Um dies zu verhindern erfolgt die Rückfrage Edwards. Wenn Sie sicher sind, daß sie die schon existieren Datei überschreiben wollen, dann wählen Sie „Ja“. Normalerweise ist es wohl sinnvoller „Nein“ zu wählen und sich die entsprechende Datei erst einmal genauer anzuschauen.

Referenz : Kapitel V. „Dateioperationen“

5) Fehler bei Betriebssystemoperationen

Meldung : **„Kein Speicher mehr!“**

Ursache : Sie leiden unter Speicherverknappung. Edward kann in diesem Fall die von Ihnen gewünschte Operation nicht korrekt ausführen.

Abhilfe : Kurzfristig ist es sinnvoll für mehr Speicher zu sorgen, indem sie nicht mehr benötigte Programme die momentan ebenfalls laufen beenden. Weiterhin können Sie Texte, die Sie nicht weiter bearbeiten möchten, mit „Frei“ aus dem „Text“-Menü wieder freigeben und hierdurch ebenfalls etwas Speicher gewinnen.
Langfristig empfiehlt sich wohl die Anschaffung einer RAM-Erweiterung.

Meldung : **„Kann Datei nicht öffnen“**

Ursache : Edward versuchte eine von Ihnen angegebene Datei zum Laden oder Speichern zu öffnen. Dieser Versuch schlug fehl. Eine mögliche Ursache wäre, daß diese Datei momentan von einem anderen Programm bearbeitet wird. Dies ist zum Beispiel möglich, wenn gerade ein Compiler diese Datei übersetzt.

Abhilfe : In oben erwähntem Fall empfiehlt es sich, zu warten, bis der Compiler seine Arbeit beendet hat und dann erneut zu versuchen, diese abzuspeichern. Ansonsten sollten Sie versuchen die Datei unter einem anderen Namen, möglicherweise auch auf einem anderen Laufwerk abzuspeichern.

Meldung : **„Kann nicht speichern“**

Ursache : Beim Schreiben einer Datei schlug irgendetwas fehl.

Abhilfe : Überprüfen Sie, daß das Laufwerk auf welchem Sie die Datei speichern wollten genügen Speicherplatz bietet und kein Schreibschutz aktiviert ist. Sollte dies nicht der Fall sein, so wechseln Sie das Laufwerk, oder deaktivieren den Schreibschutz.

Meldung : **„Kann Info-Datei nicht speichern“**

Ursache : Das Betriebssystem konnte die „.info“-Datei zu Ihrem Text nicht erzeugen.

Abhilfe : s. oben.

Meldung : **„Fehler beim Lesen“**

Ursache : Während des Lesens einer Datei trat ein Fehler auf. Möglicherweise sind Daten auf dem entsprechenden Laufwerk defekt.

Abhilfe : Versuchen Sie mit einem Programm wie DiskDoctor zu retten, was noch zu retten ist. Viel Glück!
Wenn Sie wie empfohlen fleißig Backups Ihrer Arbeiten gemacht haben, dann dürften Sie in dieser Situation nicht viel verloren haben.

Meldung : **„Konnte Datei nicht speichern“**

Ursache : Beim speichern einer Datei schlug irgendetwas fehl.

Abhilfe : s. „Kann nicht speichern“

Meldung : **„Konnte Formular nicht öffnen“**

Ursache : Edward versuchte ein von Ihnen gewünschtes Formular zu öffnen. Dies schien dem Betriebssystem nicht möglich zu sein. Normalerweise ist ein Mangel an freiem Speicher (insbesondere CHIP-Memory) an diesem Verhalten schuld.

Abhilfe : Für freien Speicher sorgen (s. „Keinen Speicher mehr“).

6) Eingabefehler

Meldung : **„Wrap-Position darf nicht kleiner Als 10 sein!“**

Ursache : Sie haben wahrscheinlich in der Kommandozeile eine Wrap-Position kleiner als 10 Zeichen setzen wollen. Da dies nicht sehr sinnvoll ist hat Edward dies unterbunden.

Abhilfe : Eine Position größer oder gleich zehn Zeichen angeben.

Referenz : Kapitel X. „Word-Wrap“

Meldung : **„Tabulatorwert nur zwischen 1 bis 20“**

Ursache : Sie haben versucht einen Wert für die Größe des Tabulatorabstandes anzugeben, der als nicht sinnvoll erachtet wird.

Abhilfe : Sinnvollen Wert zwischen Eins und Zwanzig angeben.

Referenz : Kapitel III. „Eingabe und Bearbeitung von Texten“ Abschnitt 4) RETURN, ENTER und TAB
Kapitel XVI. „Globale Einstellungen“ Abschnitt 1) „Das ‘Global Prefs’-Formular“

Meldung : **„Auflösungen nur im Bereich von 0 bis 2“**

Ursache : Sie versuchten einen ungültigen Wert für den Display-Typ anzugeben.

Abhilfe : Wert zwischen Null und Zwei angeben.

Referenz : Kapitel XVI. „Globale Einstellungen“ Abschnitt 1) „Das ‘Global Prefs’-Formular“

Meldung : **„Zeiten kleiner 30 Sek. machen keinen Sinn!“**

Ursache : Die künstliche Intelligenz Edwards hält Intervalle unter dreissig Sekunden für die „Auto-Save“-Funktion für nicht besonders sinnvoll. Wahrscheinlich haben Sie einen Wert unter dreissig Sekunden nur aus Versehen angegeben.

Abhilfe : Werte größer dreissig angeben.

Referenz : Kapitel XVI. „Globale Einstellungen“ Abschnitt 1) „Das ‘Global Prefs’-Formular“

Meldung : **„Automode-Bereich nur von 0 bis 2!“**

Ursache : Sie haben einen ungültigen Wert für den „Auto-Save“-Modus angegeben.

Abhilfe : Wert zwischen Null und Zwei angeben.

Referenz : Kapitel XVI. „Globale Einstellungen“ Abschnitt 1) „Das ‘Global Prefs’-Formular“

6) Fehler in den Definitionsdateien

Als Referenz zu diesem Abschnitt empfiehlt es sich, das Kapitel XX. „Benutzerdefinierte Menüs, Tastaturbelegung und HotKey“ zu lesen.

Meldung : **„Präprocessor-Befehl zu komplex“**

Ursache : In einer Definitons-Datei, welche die Menüs und die Tastaturbelegung Edwards enthält trat eine übermäßig komplexe Verschachtelung von „#if“-Statements auf. Da Edward hier nur eine Tiefe von maximal 16 beherrscht, hat er hier das Handtuch geworfen.

Abhilfe : Formulieren Sie ihre Definitons-Dateien etwas weniger komplex. Es erscheint schon fast unglaublich, daß Sie selbst diese Schachtelungstiefe verstehen können.

Meldung : **„#endif ohne #if“**

Ursache : Sie haben ein #endif in ihrer Datei, zu dem kein #if gehört.

Abhilfe : Überprüfen Sie die Datei noch einmal genau.

Meldung : **„Präprocessor-Befehl erwartet“**

Ursache : Sie haben in der Datei ein „#“-Zeichen an einer Stelle verwendet, an welcher der Präprozessor dieser bearbeitet. Dieser erwartet nach dem „#“-Zeichen ein Statement („if“, „endif“ oder „else“).

Abhilfe : Überprüfen Sie die Datei. Entfernen Sie das Zeichen, oder ergänzen Sie das entsprechende Statement.

Meldung : **„TITLE erwartet“**

Ursache : Sie haben in der Datei versucht ein Menü zu definieren, daß jedoch keinen einzigen Titel enthält.

Abhilfe : Die entsprechende Stelle der Datei untersuchen. Vielleicht haben Sie den „TITLE“ Befehl ja einfach nur vergessen. Sollten Sie zu den Puristen gehören, die vollständig auf Menüs verzichten können und alles mit der Tastatur machen, dann entfernen Sie auch noch den „MENU“-Befehl aus der Datei.

Meldung : **„Zu viele Menüeinträge“**

Ursache : Sie haben versucht, ein Menü mit mehr als zwanzig Einträgen zu erzeugen. Dies ist leider nicht möglich.

Abhilfe : Die Einträge auf mehrere Menüs verteilen.

Meldung : **„Zu viele Menütitel“**

Ursache : Sie haben versucht, mehr als 16 Menüs zu erzeugen. Dies ist leider nicht möglich.

Abhilfe : Beschränken Sie sich etwas in Ihren Menüs.

7) Fehler beim Analysieren von Kommandos

Fehlermeldungen in Definitions-Dateien oder in Kommandozeilen folgt immer eine Angaben der Position, an welcher der entsprechende Fehler auftrat. Die Fehlermeldungen haben daher folgenden Form:

1. **Fehler**
   In Zeile : Zeile x Column: Spalte

oder

2. **Fehler**
   In Spalte: Spalte

Meldung : **„IDENTIFIER erwartet“**

Ursache : Es wurde ein IDENTIFIER erwartet, aber nicht gefunden.
Ein IDENTIFIER beginnt mit einem Buchstaben, gefolgt von einer beliebigen Anzahl alphanumerischer Zeichen (also Buchstaben und Ziffern) gemischt mit dem Unterstrich „_“.

Meldung : **„STRING erwartet“**

Ursache : Es wurde ein STRING erwartet. Ein String beginnt mit einem Hochkomma („) gefolgt von einer beliebigen Anzahl Zeichen und wird mit einem Hochkomma abgeschlossen.

Meldung : **„NUMBER erwartet“**

Ursache : An dieser Stelle wurde eine Zahl erwartet.

Meldung : **„(erwartet“**

Ursache : Es wurde eine öffnende, runde Klammer erwartet.

Meldung : **„) erwartet“**

Ursache : Eine schliessende, runde Klammer wurde erwartet.

Meldung : **„; erwartet“**

Ursache : Es wurde eine Semikolon erwartet.

Meldung : **„Kein Befehl“**

Ursache : Es wurde ein Kommando aus Edwards Befehlsvorrat erwartet.

Abhilfe : Überprüfen Sie die Stelle. Wahrscheinlich haben Sie den Befehl nur falsch ausgeschrieben.

Meldung : **„LIST oder IDENTIFIER erwartet“**

Ursache : Es wurde eine LIST oder ein IDENTIFIER erwartet, aber nicht gefunden. Eine LIST wird von runden Klammern umschlossen. Zwischen diesen können sich IDENTIFIER, STRINGs, NUMBERs, COMMANDs oder LISTs auch gemischt befinden.

Meldung : **„LOBJECT erwartet“**

Ursache : Es wurde ein LOBJECT erwartet. Ein LOBJECT kann ein STRING, eine NUMBER, ein COMMAND oder eine LIST sein.

8) Fehlermeldungen beim Auswerten von Kommandos

Meldung : **„LVAL erwartet“**

Ursache : Ein LVAL, das bedeutet eine Zahl wurde von einem Kommando als Argument erwartet.

Meldung : **„Zu viele Argumente“**

Ursache : Ein Kommando wurde mit zuvielen Argumenten versorgt.

Meldung : **„Falsches Argument“**

Ursache : Ein Kommando wurde mit falschen Argumenten versorgt.

Sekundärliteratur

- „Das kleine Arschloch“ von Walter Moers, Eichborn Verlag
  Eine Sammlung abgeschmackter Comic-Strips, in denen kein Tabu unberührt bleibt. Hardcore-Humor vom Feinsten. Erklärt außerdem die merkwürdige Vorliebe des Autors dieses Handbuchs für das Wort „Mahlzeit“.

- „Das kleine Arschloch kehrt zurück!“ von Walter Moers, Eichborn Verlag
  Genauso abgeschmackt wie das vorige Buch. Sehr empfehlenswert. „Peppi, peppi, peppi“.

- Das Telefonbuch von Los Angeles.
  Die Telefonnummern der Stars. Rufen Sie z.B. einmal bei Karl Malden an, dem Star aus die „Strassen von San Francisco“. Auch sonst viele interessante Informationen. Sehr entspannende Lektüre.

- „Schlecht ist uns nicht schlecht genug“, von Thomas E. Wieger und Jörg W. Schmidt.
  In mühevoller Kleinarbeit recherchierten die Autoren nach GAGs, den größen anzunehmenden Geschmacklosigkeiten. Alles über die schlechtesten Trash-Fernsehserien der Welt, die ekligsten Nahrungsmittel, die grauenhaftesten Kleidungsstücke, die abstossendsten Schauspieler u.v.a.m..

Epilog

Sehr geehrte Leserin, verehrter Leser,
wer weiß, ob Sie jemals am Ende dieses Handbuches angelangen. Hätte ich zwar eigentlich lieber einen Roman geschrieben, als dieses Handbuch, so hoffe ich doch, daß meine Ausführungen Ihnen die mächtigen Fähigkeiten und Features von Edward nahegebracht haben. Ich bedanke mich für Ihre Aufmerksamkeit und wünsche Ihnen noch viel Erfolg und eine immens gesteigerte Produktivität durch den Einsatz Edwards.
Mahlzeit!

Index

A

Absatz
  neu umbrechen 36
ARexx 61, 65
Auszeichnen 32
Auto-Save 50

B

BACKSPACE 12
Block
  Drucken 23
  Einrücken 23
  Groß- bzw. Kleinbuchstaben 24
  Laden 23
  markieren 20
  Speichern 23
  Umwandeln 24
Blockoperationen 20

C

CLI 10
Copy 20
Cursor-Tasten 13
Cut 20

D

Default-Einstellungen 49
Definition 55
DELETE 12

E

EBNF-Darstellung 53
Edit-Prefs Formular 34
Editor
  Arbeit Unterbrechen 44
  Verlassen 44
EdwardDefault.info 93
EMPTY 56
ENTER 12
Error-Datei 28

F

Falten
auszeichnen ... 33
Entferne ... 32
Entferne alle ... 32
erzeugen ... 31
gefalteten Zeilen anspringen ... 33
rückgängig ... 32
schachteln ... 32
Farben ... 48
Fehlermeldungen
#endif ohne #if ... 104
(erwartet ... 105
) erwartet ... 105
; erwartet ... 106
Alles ist rückgängig gemacht ... 96
Alles ist wiederhergestellt ... 96
Altes Pictogramm ersetzen ... 98
Auflösungen nur im Bereich von 0 bis 2 ... 102
Automode-Bereich nur von 0 bis 2 ... 103
Cursor auf keinem Strukturelement ... 95
Datei existiert bereits ... 99
Falsches Argument ... 106
Fehler beim Lesen ... 101
Geänderten Text verwerfen ... 97
IDENTIFIER erwartet ... 105
Kann Datei nicht öffnen ... 100
Kann Info-Datei nicht speichern ... 101
Kann nicht speichern ... 100
Kein Befehl ... 106
Kein Block definiert ... 94
Kein Pictogramm vorhanden ... 93
Kein Speicher mehr ... 97, 100
Konnte Backup nicht erzeugen ... 93
Konnte Formular nicht öffnen ... 101
LIST oder IDENTIFIER erwartet ... 106
LOBJECT erwartet ... 106
LVAL erwartet ... 106
Markierung gesetzt ... 92
Markierung nicht gesetzt ... 94
Menü ist zu groß ... 96
NUMBER erwartet ... 105
Präprocessor-Befehl erwartet ... 104
Präprocessor-Befehl zu komplex ... 103
STRING erwartet ... 105
Strukturen stimmen nicht überein ... 96
Suchmuster nicht gefunden ... 95
Tabulatorwert nur zwischen 1 bis 20 ... 102
Text ist geschützt ... 93
TITLE erwartet ... 104

Wrap-Position fehlerhaft 102
würde Einfaltung zerstören 94
Zeichenkette zu lang 95
Zeiten kleiner 30 Sek. machen keinen Sinn 103
Zu viele Argumente 106
Zu viele Menüeinträge 104
Zu viele Menütitel 104
Fenster
Erzeugen 40
horizontal teilen 43
mehreren Ansichten 42
Schließen 42
vertikal teilen 43
volle Größe 43
Wahl 41
Find"-Formular 25
Fließtext 36
Fußzeile 51

G

Globale Einstellungen 48
Grenzen 63

H

HotKey 46

I

Iconify 46
Installation 9
interpretieren 28
ITEM 56

K

KeyDef 58
Keys 58
Klammer-Check 27
Kommandos 47, 64
Again 74
AutoMode 85
BackSpace 65
BigStepDown 66
BigStepUp 66
BlockHigher 73
BlockLower 72
BlockSL 72
BlockSR 72
Bot 68
Copy 71
Cut 71

Delete 65
Display 85
Do 85
EditForm 84
EndTextRead 90
EnterCmd 84
ExecCli 91
ExecRexx 86
ExecRexxString 86
Exit 82
Find 74
FindForm 83
FindRepForm 83
Fold 76
Font 85
FreeText 68
FreeWin 81
FullSizeWin 82
GetBlockInfo 88
GetLineString 87
GetScreenInfo 89
GetTextInfo 88
GetTextList 88
GetTextPos 87
GetWindowList 88
GetWord 87
GlobalForm 83
GoBookMark 77
GoDispPos 67
GoMatchStruct 75
GoParsedError 75
GoPredFold 76
GoPredKey 77
GoStringPos 67
GoSuccFold 77
GoSuccKey 77
GotoForm 83
Iconify 91
InfoForm 84
InitTextRead 90
InsertChars 91
InsertFile 70
Jump 75
KillLine 65
Mark 71
MarkVert 71
ModifyOldCmd 84
MoveDown 66
MoveLeft 67
MoveRight 67
MoveUp 66

New 69
NewText 68
NewWin 81
Open 70
OpenClip 72
OpenDefs 86
PageForm 83
Paste 71
PasteVert 71
PlayRecScript 80
PredText 68
PredWin 81
Print 69
Quit 82
RecPlay 80
RecStart 80
RecStop 80
ReDo 91
RemFold 76
RemTab 74
RepeatCmd 84
Replace 74
ReplaceAll 74
Return 66
Save 70
SaveClip 72
SaveRecScript 80
ScreenDown 67
ScreenUp 67
SelTextN 69
SetBookMark 77
SetFoldsOff 76
SetFoldsOn 76
SetMode1, SetMode2, SetMode3 91
SetOverOff 78
SetOverOn 78
SetTab 78
SetWrap 78
SetWrapOff 78
SetWrapOn 78
ShowFold 76
SplitHorWin 81
SplitVerWin 81
SuccText 69
SuccWin 81
TextRead 90
ToggleKey 77
Top 68
UnDo 90
UndoLine 65
UpdateText 70

UpdateTexts 70
WrapPara 79
WrapText 79
Kommandozeile
Eingabe 47
erneut ausführen 47
letzte Eingabe 47
Kommandozeilen-Interpreter 47

L

Lesezeichen 33
LIST 64

M

Makros
abspielen 38
ARexx 39
aufzeichnen 37
speichern 38
MENU 56
Menü-Gruppe 56
MenuItem 56
ModeQualList 58

O

Open Defs 61
OPT_ 64
Overstrike 34

P

Parameter 64
Paste 20
Peppi 107
Pictogramm 18, 49
positionieren
Anfang 29
Ende 29

Q

Qualifier 57
QuotedString 55

R

Redo 62
RETURN 12
Right Margin 35
Rückgabewert 64

S

Schieberegler 14
Seitenformat 51
Seitennummer 51
SMALL_STRING 64
Sondertaste 58
Spaces 35
Spaltenblöcke 21
Speichern
  Texte 18
  Zusatzinformationen 18
Standard-Datei 93
Status-Anzeige 11
STRING 64
Struktur-Check 27
Suchen
  Groß-/Kleinbuchstaben 26
  vollständige Wörter 26
Suchmuster 26
Suchrichtung 26

T

TAB 12
Tab Width 35
Tabs 35
Tastaturbelegung 56
Text
  Drucken 17
  Erzeugen 15
  Freigeben 16
  Laden 19
  Löschen 17
  neu umbrechen 36
  Speichern 18
  Wahl 16
TITLE 56

U

Umbruchposition 36
Umwandeln 24
Undo 62
Undo-Depth 49
Undo-Funktion 49, 63
Undo-Tiefe 64
UndoLine 12
UndoMem 49
Unterbrechen 44

V

VALUE 64
Verlassen 45
vertauschen 27
Voreinstellungen 34

W

Word-Wrap 34, 36, 50
Workbench 9

Z

Zeichenfolge 27
Zeichensatz 49
Zeilen positionieren 29
Ziffernblock 13
Zusatzinformationen 50
Zwischenspeichern 21

Maxon C++

Tutorial

MAXON
computer

Inhaltsverzeichnis

Einführung in C++

Ein paar kurze Bemerkungen, bevor es los geht 9

1. Der Einstieg 10

1.1 Ein erstes Beispiel 10

1.2 Lexikalisches 12

1.3 Zahlen, Variablen und Berechnungen 15

1.4 Numerische Datentypen und Operatoren 17
1.4.1 Ganzzahlige Datentypen 17
1.4.2 Literale für ganze Zahlen und Zeichenketten 19
1.4.3 Ganzzahlige Operatoren und Berechnungen 22
1.4.3.1 Allgemeines 22
1.4.3.2 Postfix-Operatoren 22
1.4.3.3 Unäre Operatoren 23
1.4.3.4 Multiplikative Operatoren 25
1.4.3.5 Addition und Subtraktion 26
1.4.3.6 Shift-Operatoren 27
1.4.3.7 Vergleiche 28
1.4.3.8 Bitweise Verknüpfungen 29
1.4.3.9 Logische Operatoren 29
1.4.3.10 Der bedingte Ausdruck 30
1.4.3.11 Zuweisungen 32
1.4.3.12 Der Komma-Operator 33
1.4.4 Fließkommatypen 33
1.4.4.1 Literale und Typen 33
1.4.4.2 Interne Darstellung der Daten 34
1.4.4.2 Fließkommaoperationen 35
1.4.5 Auswertung von Ausdrücken 35
1.4.6 Explizite Typumwandlungen 38
1.4.7 Initialisierungen 39

1.5 Anweisungen 40
1.5.1 Bedingungen und Verzweigungen 40
1.5.2 Anweisungen, Deklarationen und Blöcke 43
1.5.3 Schleifen 46
1.5.3.1 Die „while"-Anweisung 46
1.5.3.2 Die „for"-Schleife 47
1.5.3.3 Die „do"-Schleife 49

1.5.4 Sprünge 50
1.5.4.1 Sinn und Unsinn von „goto“ 50
1.5.4.2 „break“ und „continue“ 52
1.5.5 Vielfachverzweigungen 54
1.5.6 Die „return“-Anweisung 57

1.6 Funktionen 57
1.6.1 „main“ allein genügt nicht 57
1.6.2 Parameter und Argumente 58
1.6.3 Rückgabe von Werten 60
1.6.4 Funktionen und Gültigkeitsbereiche 64
1.6.4.1 Deklarationen auf Dateiebene 64
1.6.4.2 Rekursive Funktionen 65
1.6.4.3 Deklarationen und Definitionen 66
1.6.5 Überladen, Default-Parameter und andere Spezialitäten 68
1.6.5.1 Überladen von Funktionen 68
1.6.5.2 Abkürzen durch Default-Argumente 71
1.6.5.3 Variable Parameterlisten 73
1.6.6 Anachronismen aus der Mottenkiste 74
1.6.6.1 Umschalten in den C-Modus 74
1.6.6.2 Parameterlisten im alten Stil 75
1.6.6.3 Von der Steinzeit in die ANSI-Ära 77
1.6.7 Einige Standardfunktionen 77
1.6.7.1 Allgemeines 77
1.6.7.2 Ein- und Ausgabe: <stdio.h> 78
1.6.7.3 Zeichen, Zeichenketten und Bytefolgen: 79
1.6.7.4 Fließkommafunktionen: <math.h> 80
1.6.7.5 Die C-Fundgrube: <stdlib.h> 80

2. Datentypen und Deklarationen

2.1 Vektoren, Zeiger und Referenzen 83
2.1.1 Pointer 83
2.1.1.1 Von Objekten und Adressen 83
2.1.1.2 Zeiger: Deklaration und elementare Operationen 84
2.1.1.3 Zeiger als Funktionsparameter 85
2.1.1.4 Zeigertypen 87
2.1.1.5 Was man sonst noch so mit Zeigern machen kann 89
2.1.2 Vektoren 90
2.1.2.1 Das Vektor-Konzept in C 90
2.1.2.2 Zeiger und Vektoren 92
2.1.2.3 Genaueres über Vektordeklarationen 95
2.1.2.4 Vektoren als Parameter 97
2.1.2.5 Strings 98
2.1.2.5.1 Konstante Zeichenketten 98
2.1.2.5.2 Initialisieren, Kopieren und Zuweisen 100
2.1.2.5.3 Stringvergleiche 103
2.1.2.5.4 Argumente aus der Kommandozeile 107
2.1.2.6 Initialisierung von Vektoren 108
2.1.3 Referenzen 111

2.1.3.1 Einführung 111
2.1.3.2 „VAR-Parameter" und temporäre Objekte 112
2.1.3.3 Weitere Möglichkeiten und Einschränkungen 114

2.2 Qualifizierungen, Deklarationen und Spezifikationen 116
2.2.1 Speicherklassen 116
2.2.1.1 Automatische und statische Daten 116
2.2.1.2 Module und ihre Gültigkeitsbereiche 118
2.2.1.2.1 Gemeinsame Daten und Funktionen 118
2.2.1.2.2 Modulinterne „static"-Namen 121
2.2.1.2.3 Typsicheres Linken und Link-Spezifiationen 122
2.2.1.2.4 Die Schnittstelle zum Amiga-Betriebssystem 125
2.2.1.3 Daten in Registern 126
2.2.1.4 Daten im CHIP-RAM 128
2.2.2 Typdefinitionen 129
2.2.3 Konstante Daten auf der Flucht 131
2.2.3.1 Die „const"-Qualifizierung 131
2.2.3.2 Flüchtige Daten: „volatile" 134
2.2.4 Inline-Funktionen 136

2.3 Zeiger auf Funktionen 138
2.3.1 Wohin zeigen Zeiger auf Funktionen? 138
2.3.2 Noch ein paar Anmerkungen zu Syntax und anderen Details 139
2.3.3 Zeiger auf Überladene Funktionen 140
2.3.4 Eine Anwendung: „qsort" und „bsearch" 141

2.4 Aufzählungen 144
2.4.1 Deklaration 144
2.4.2 Typnamen und Typlabel 146
2.4.3 Aufzählungen und Zahlen 147

2.5 Strukturen 150
2.5.1 Grundlagen 150
2.5.2 Definitionen und Deklarationen 152
2.5.3 Initialisierung 155
2.5.4 Gültigkeitsbereiche und Scope-Regeln 156
2.5.5 Unionen 159
2.5.6 Bitfelder 162

2.6 Dynamisch organisierte Datenstrukturen 164
2.6.1 Die Idee 164
2.6.2 „new" und „delete" 165
2.6.3 Dynamisch erzeugte Vektoren 166
2.6.4 Initialisierungen 168
2.6.5 „malloc" und andere Antiquitäten 169
2.6.6 Beispielprogramm: Eine Liste von Daten 171
2.6.6.1 Das Problem: Ein wirrer Haufen von Daten 171
2.6.6.2 Die Lösung: Eine doppelt verkettete Liste 173
2.6.6.3 Initialisieren und Einfügen 175
2.6.6.4 Suchen und Löschen 179
2.6.6.5 Das Programm an einem Stück 181

3. Objektorientiertes Programmieren

3.1 Klassen und Objekte 187
3.1.1 Keine Panik! 187
3.1.2 Member-Funktionen 189
3.1.2.1 Erste Beispiele 189
3.1.2.2 Member-Funktionen (Methoden) und konstante Objekte 191
3.1.2.3 Inlining von Member-Funktionen 192
3.1.3 Klassen und Zugriffsrechte 193
3.1.4 No Scope, no hope! 196
3.1.5 Friends 197
3.1.6 Statische Member 200

3.2 Vererbungslehre 203
3.2.1 Abgeleitete Klassen 203
3.2.1.1 Ein erstes Beispiel 203
3.2.1.2 Mehrfach-Vererbung und andere Features 206
3.2.1.3 Vererbung und Typregeln 209
3.2.2 Scoperegeln und Zugriffskontrolle bei Vererbung 211
3.2.2.1 Noch mehr Details 211
3.2.2.2 Der Zugriffsmodus „protected" 212
3.2.2.3 Zugriffs-Deklarationen 213
3.2.3 Virtuelle Member-Funktionen 215
3.2.3.1 Ein Beispiel 215
3.2.3.2 Die Details 217
3.2.3.3 Abstrakte Klassen 220
3.2.4 Ein paar Anmerkungen zur Implementation 221
3.2.5 Virtuelle Basisklassen 225

3.3 Konstruktoren und Destruktoren 234
3.3.1 Initialisierung von Klassen 234
3.3.2 Konstruktoren mit Argumenten 236
3.3.3 Initialisierung von Basisklassen und Membern 238
3.3.4 Besondere Konstruktoren 242
3.3.4.1 Default-Konstruktoren 242
3.3.4.2 Kopier-Konstruktoren 243
3.3.5 Destruktoren 246

3.4 Beispielprogramm: 250

3.5 Noch mehr Features 266
3.5.1 Zeiger auf Member 266
3.5.2 Konvertierungsfuntionen 270
3.5.3 Temporäre Objekte 273

4. Überladen

4.1 Überladene Funktionen 277
4.1.1 Was man darf, und was nicht 277

4.1.2 Was der Compiler davon hält 278
4.1.2.1 Die Hierarchie der Typwandlungen 278
4.1.2.2 Matching von Funktionsargumenten 284

4.2 Operatoren 286
4.2.1 Allgemeine Operatoren 286
4.2.1.1 Allgemeines über allgemeine Operatoren - allgemeiner geht's kaum 286
4.2.1.2 Wie man so etwas deklariert 288
4.2.1.3 Und wie man es benutzt 289
4.2.2 Spezielle Operatoren 291
4.2.2.1 Der Zuweisungs-Operator „=“ 291
4.2.2.2 „new“ und „delete“ 294
4.2.2.2.1 Selbstgeschnitzte Speicherverwaltung 294
4.2.2.2.2 Überladen von „new“ 295
4.2.2.2.3 Der „delete“-Operator 298
4.2.2.3 Funktionsaufruf als Operator 301
4.2.2.4 Klassen als Möchtegern-Vektoren 303
4.2.2.5 Intelligente Zeiger: "->" und das unäre ""* 305
4.2.2.6 Inkrement und Dekrement 306
4.2.2.7 Ein- und Ausgabe 308
4.2.3 Beispielprogramm: Die „string“-Bibliothek 310

5. Der Preprozessor

5.1 Allgemeines 319

5.2 Was Sie schon immer über #define wissen wollten, aber nicht zu fragen wagten 320
5.2.1 Makrodefinition 320
5.2.2 Makros mit Parametern 323

5.3 Was Sie noch nie über #include wissen wollten, aber jetzt trotzdem erfahren 325

5.4 Bedingte Übersetzung 326

5.5 Pragmatismus 329
5.5.1 Allgemeines und Wiederholungen 329
5.5.2 Breakpoint-Kontrolle 329
5.5.3 Amiga-spezifische Systemaufrufe 331

5.6 Noch mehr Features 333
5.6.1 Verbinden von Zeilen 333
5.6.2 Verbinden konstanter Zeichenketten 333
5.6.3 Trigraph-Sequenzen 334
5.6.4 Hausgemachte Fehlermeldungen 335
5.6.5 Getürkte Dateinamen und Zeilennummern 336
5.6.6 Nix dahinter 336

5.7 Vordefinierte Symbole 337

5.8 Arbeitsweise des Preprozessors 339

6. Was Sie als Amiga-Programmierer wissen sollten

6.1 Die Schnittstelle zum Betriebssystem 341

6.2 Start von der Workbench 342

Index 347

Einführung in C++

Ein paar kurze Bemerkungen, bevor es los geht

Dieses Buch soll eine Einführung in das Programmieren mit MaxonC++ darstellen. Es wäre sehr nützlich, wenn Sie bereits eine Programmiersprache beherrschen, und wenn es bloß AmigaBASIC ist. Sonst werde ich nichts als bekannt voraussetzen. Wenn Sie sogar schon C kennen, werden Sie den Anfang dieses Handbuchs wahrscheinlich nur überfliegen, denn C++ basiert auf C, und wir werden uns zunächst mit ziemlich elementaren Dingen beschäftigen, bei denen es kaum Unterschiede zwischen C und C++ gibt.

Damit wären wir auch schon bei einem kleinen Problem: Eigentlich ist dies eine Einführung in gleich drei Programmiersprachen: C, C++ und MaxonC++. Der Grund dafür liegt in dem Chaos, das es bei den diversen Sprachstandards gibt: Da wäre zum einen der ANSI-C-Standard, den MaxonC++ vollständig enthält. C++ ist von ANSI C abgeleitet (genaugenommen ist es noch etwas komplizierter: C++ entstand aus der „alten" C-Sprachdefinition und ANSI C enthält außer jenem Standard einige Features aus C++, was aber nichts an der Tatsache ändert, daß ANSI C im wesentlichen eine Untermenge von C++ ist). Einen verbindlichen ANSI-Standard für C++ gibt es derzeit noch nicht, nur den de-fakto-Standard, der vom C++-Erfinder Bjarne Stroustrup festgelegt wurde und auch „AT&T"-Standard genannt wird. Davon gibt es aber verschiedene Versionen, von denen eine als „AT&T 2.0" bzw. „2.1" (das ist im Prinzip dasselbe) bekannt ist. Darauf basiert MaxonC++.

Jeder Sprachstandard enthält Dinge, die implementationsabhängig sind, d. h. der Compilerbauer hat gewisse Freiheiten. Dies ist meist durch die verschiedenen Rechnerarchitekturen erforderlich. Deshalb muß dieses Handbuch auch Auskunft darüber geben, wie die jeweiligen Details in MaxonC++ implementiert sind. Außerdem hat MaxonC++, wie wohl jeder Compiler, kleinere Features, die über die Sprachdefinition hinausgehen. Auch diese müssen im Handbuch natürlich dokumentiert werden.

Wenn ich also im folgenden von „C" oder „ANSI C" spreche, so gilt das gesagte jeweils auch für C++ und somit für MaxonC++, sofern ich mich nicht ausdrücklich auf die Unterschiede zwischen diesen drei Programmiersprachen beziehe.

1. Der Einstieg

1.1 Ein erstes Beispiel

An dieser Stelle könnte ich viele Worte über das objektorientierte Programmieren und andere wichtige und nützliche Konzepte von C++ und die Vorzüge und die Leistungsfähigkeit dieser Programmiersprache im allgemeinen und von MaxonC++ im besonderen verlieren. Entweder wissen Sie schon, was „objektorientiert" heißt, und dann würden Sie diesen Abschnitt sowieso überlesen, oder Sie kennen es noch nicht, und dann würde ein allgemeiner Exkurs zu diesem Thema bei Ihnen erfahrungsgemäß doch eher Ratlosigkeit und Verwirrung als Klarheit und Erkenntnis schaffen.

Und so ziehe ich es vor, ohne weitere Vorrede gleich zur Einführung in die Programmiersprache C++ überzugehen, und verspreche Ihnen, Sie im folgenden mit Satzungetümen wie den obigen zu verschonen. Also gleich das erste Beispiel:

```
// Das unvermeidliche Hallo-Programm
#include <stream.h>

void main()
{ cout << "Hallo!";
}
```

Am besten tippen Sie das Programm in Editor ab, starten den Compiler und lassen das Programm dann laufen, so wie es im ersten Teil dieses Handbuchs erklärt wurde. Dann sollte das Programm den Test „Hallo!" ausgeben.

Falls das Compilieren oder Linken nicht geklappt haben sollte, kann das unterschiedliche Ursachen haben. Zunächst könnte natürlich der Compiler einen Fehler melden, falls Sie das Programm nicht richtig abgetippt haben. Auch kann es sein, daß der Compiler die Includedatei „stream.h" nicht findet, weil Sie das C++-System nicht korrekt installiert haben, oder daß der Linker aus demselben Grund eine Library nicht finden kann. Dann sollten Sie sich den entsprechenden Abschnitt dieses Handbuchs noch einmal ansehen.

Nehmen wir also an, daß Sie das Prograämmchen zum Laufen gebracht haben. Was bedeutet das Ganze nun?

Die erste Zeile ist ein Kommentar. Der Compiler erkennt das an dem doppelten Slash „//" am Zeilenanfang. Ganz allgemein können Sie auch an jede Zeile eines C++-Programms auf diese Weise einen Kommentar anhängen. Der Compiler ignoriert diese Kommentare vollständig, sie dienen nur dazu, das Programm dem menschlichen Leser - also z. B. Ihnen - zu erläutern.

Es gibt noch eine andere Schreibweise für Kommentare, die eine Art Erbschaft von der Programmiersprache C ist:

```
/* Auch dies ist ein Kommentar. Er beginnt mit einem "/"
und einem "*" und kann sich dann über beliebig viele
Zeilen  erstrecken. Beendet wird er dann mit diesen beiden
Zeichen: */
```

Diese Kommentare enden also nicht automatisch am Zeilenende, sondern werden mit der Zeichenfolge „*/“ abgeschlossen. Sie sind eigentlich nur dann praktisch, wenn man einen längeren Text als Kommentar in den Programmtext setzen will. In der Praxis hat man es aber meist mit kürzeren Kommentaren zu tun, und da ist die „//“-Schreibweise einfach bequemer. Außerdem kann man daran auf den ersten Blick erkennen, daß man ein C++- und nicht etwa ein C-Programm vor sich hat. Selbst hartnäckige Gegner der objektorientierten Programmierung können nicht leugnen, daß diese Kommentare eine enorm sinnvolle Neuerung von C++ sind.

Die zweite Zeile beginnt mit einem „#“. Solche Zeilen werden genaugenommen nicht vom Compiler selbst, sondern vom sog. Preprozessor bearbeitet. Dieser Preprozessor ist ein Programm, durch das der Quelltext noch vor dem eigentlichen Compilieren läuft. Neben vielen anderen nützlichen Funktionen, auf die ich in Kapitel 5 eingehen werde, entfernt er Kommentare und führt Zeilen aus, die mit eben jenem „#“ beginnen. Das Kommando „include“ weist ihn an, an Stelle dieser Zeile den Inhalt einer Datei in den Quelltext einzufügen - hier ist es eine Datei namens „stream.h“.

Natürlich löscht der Preprozessor die Kommentare nicht wirklich aus ihrem Quelltext, und er fügt Include-Dateien auch nicht tatsächlich in den Sourcecode ein. Damit ist gemeint, daß er eine entsprechend modifizierte Kopie des Quelltexts erzeugt, die dann vom Compiler übersetzt wird. In Wirklichkeit wird in MaxonC++ gar keine echte Kopie angelegt (das würde nur Speicher verschwenden), aber lassen wir das und tun einfach so als ob.

Die Datei „stream.h“ enthält einige Definitionen, die für die Ein- und Ausgabe nahezu unverzichtbar sind. Deshalb werden Sie diese Include-Datei wohl in den meisten Programmen verwenden.

Nach dem Kommentar für den Leser und der Preprozessor-Anweisung beginnt nun unser eigentliches Programm.

Die Zeile

```
void main()
```

bedeutet folgendes:

- Es wird eine Funktion definiert (gekennzeichnet durch das leere Klammerpaar „()“).
- Der Name dieser Funktion ist „main“. Das ist eine ganz besondere Funktion, die in keinem Programm fehlen darf: Sie stellt das Hauptprogramm dar.
- Die Funktion hat keinen Rückgabewert, bzw. einen Wert des leeren Typs „void“. Was das bedeutet, werden Sie bei passender Gelegenheit erfahren.

Als nächstes folgt der Inhalt oder „Rumpf“ dieser Funktion. Er ist mit geschweiften Klammern „{“ und „}“ einzuschließen und enthält hier nur eine einzige Anweisung:

```
cout « "Hallo!";
```

Das Wort „cout“ bezeichnet ein Datenobjekt, das in der Includedatei „stream.h“ deklariert wird - deshalb brauchten wir also die „#include“-Zeile. Was ein Datenobjekt im allgemeinen ist und wie das Objekt „cout“ genau aussieht und ob ein Objekt wirklich irgendwie „aussieht“ oder ob das nur so eine Redensart ist und viele Fragen mehr, möchte ich an dieser Stelle erst einmal zurückstellen. Sie sollten sich zunächst einmal damit begnügen, daß man in C++ mit einer Anweisung der Art

```
cout « Ausdruck;
```

einen Ausdruck auf den Bildschirm ausgibt, in unserem Fall war der Ausdruck der Text „Hallo!“. Jede Anweisung endet in C++ mit einem Semikolon.

1.2 Lexikalisches

An dieser Stelle sind noch einige ganz allgemeine Anmerkungen über die Gestalt von Programmen in C++ nötig:

Zunächst ist zu erwähnen, daß in C++ die Groß-/Kleinschreibung von Namen generell signifikant ist. Mit der Zeile

```
Void MAIN()
```

hätte der Compiler also nichts anfangen können. Hier unterscheidet sich C++ von Programmiersprachen wie Pascal oder dem allseits beliebten AmigaBasic.

Als Ausgleich für diese Pingeligkeit ist der Compiler aber in bezug auf die Textformatierung absolut tolerant. Es ist ihm ziemlich egal, wie Sie Ihr Programm in Zeilen aufteilen, wie weit Sie Zeilen einrücken, wo Sie Leerzeichen oder Kommentare einfügen usw. Unsere Funktion „main" hätten Sie also auch so schreiben können:

```
void main (
 ){/* Dies ist ein Kommentar */cout
« /* */ "Hallo!" // Noch ein Kommentar
; } // Hier ist das Programm zuende. Grüße an Oma!
```

...oder so:

```
void main(){cout«"Hallo!";}
```

Für Menschen wie Sie und mich ist das natürlich nicht so besonders leserlich, aber der Compiler betrachtet ein Programm ausschließlich als eine Folge von Eingabesymbolen, deren textliche Anordnung keine Rolle spielt. Wichtig ist nur, daß er die Symbole voneinander trennen kann, z. B. ist deshalb eine Lücke zwischen „void" und „main" unentbehrlich. Im Gegensatz dazu achtet der Preprozessor durchaus auf die richtige Zeilenaufteilung: Eine Preprozessor-Anweisung beginnt stets mit einem „#", das in der ersten Spalte einer Zeile stehen muß, und ist immer genau eine Zeile lang.

Da wir gerade beim Thema sind, können wir auch gleich die unterschiedlichen Symbole betrachten, aus denen sich ein Programm zusammensetzt.

Zunächst wären da die Wortsymbole. In unserem kleinen Beispielprogramm benutzen wir nur das Wortsymbol (oder Schlüsselwort) „void". Eine vollständige Übersicht über alle Schlüsselwörter von C++ finden Sie im Anhang dieses Handbuchs.

Etwas scheinbar Ähnliches, aber in Wahrheit ganz anderes sind die Bezeichner (Identifier). Das sind vom Benutzer definiertbare Namen für Objekte, Funktionen, Datentypen und einiges mehr. Im „Hallo"- Programm benutzen wir den Bezeichner „cout", der in „stream.h" deklariert wurde, und definieren selbst den Bezeichner „main" als Namen einer Funktion.

Ein Bezeichner ist eine beliebig lange Folge von Buchstaben, Ziffern und dem Unterstrich „_". Es darf aber keine Ziffer am Anfang stehen. Gültige Bezeichner wären also z. B.

```
main
TeSt
Eumel26731
Dies_ist_ein_ziemlich_langer_Name
_1_
x
```

Außerdem gibt es diverse Literale, das sind Konstanten mit einem bestimmten Wert. Wir haben bisher nur eine Zeichenketten-Konstante benutzt, nämlich „Hallo!“. Eine solche Zeichenkette (String) wird mit doppelten Anführungszeichen eingeschlossen und kann beliebig viele Zeichen enthalten. Sie darf aber ein Zeilenende nicht überschreiten. Wenn man mehrere Zeichenketten hintereinander schreibt, hängt der Preprozessor sie aneinander, so daß die Anweisung

```
cout « "Dies" "sind" "mehrere Zeichenketten"".";
```

identisch mit

```
cout « "Diessindmehrere Zeichenketten.";
```

ist.

Man kann mit dem Backslash „\“ besondere Zeichen in einen solchen String einschließen Hier sollen nur zwei erwähnt werden: Die Folge „\n“ wird in einem String durch das Linefeed-Zeichen ersetzt, bewirkt also bei der Ausgabe einen Zeilenvorschub.
Ein Beispiel:

```
cout « "Life! Don't talk\nto me about life!";
```

gibt

```
Life! Don't talk
to me about life!
```

aus. Sie können doppelte Anführungszeichen in einen String einschließen, indem Sie einen Backslash voranstellen:

```
cout « "Written by Jens \"Himpelsoft\" Gelhar 1992.";
```

schreibt

```
Written by Jens "Himpelsoft" Gelhar 1992.
```

Außerdem gibt es noch etliche Literal-Schreibweisen für Zahlen, die wir im nächsten Kapitel kennenlernen werden.

Die vierte Kategorie von Eingabesymbolen sind die Sonderzeichen: Diverse eckige, runde und geschweifte Klammern, Operatoren wie z. B. „«“, Trenner wird das Semikolon „;“ oder das Komma „,“ und einige andere, die wir noch kennenlernen werden. Auch dazu gibt es eine vollständige Übersicht im Anhang.

1.3 Zahlen, Variablen und Berechnungen

Nun können Sie also schon per Programm Texte ausgeben. Bekanntlich kann ein Computer aber wesentlich besser mit Zahlen umgehen, und genau darum soll es in diesem Kapitel gehen.

Außerdem benötigt man zum Programmieren auf jeden Fall Variablen. Eine Variable ist in C, genau wie in allen anderen real existierenden Programmiersprachen auch, ein Datenobjekt eines festgelegten Typs, das unterschiedliche Werte enthalten kann. Man definiert einfache Variablen in C nach folgendem Schema:

```
Typname Variablenname, Variablenname, ..., Variablenname;
```

Am besten betrachten wir gleich ein Beispielprogramm, im dem ein paar Variablen und andere neue Sprachelemente vorkommen:

```
#include <stream.h>

void main()
{ // Variablen muß man definieren, bevor man sie benutzt:
  int Eingabe1, Eingabe2;
  int Summe, Produkt, Mittelwert;

  // Jetzt kommen die Anweisungen unseres Programms:
  cout « "Bitte geben Sie zwei Zahlen ein: ";

  // Die folgende Anweisung erwartet zwei Eingaben des Benutzers:
  cin » Eingabe1 » Eingabe2;

  // Nun haben diese beiden Variablen Werte, und wir können
  // einige Berechnungen durchführen:
  Summe = Eingabe1+Eingabe2;
  Produkt = Eingabe1*Eingabe2;
  Mittelwert = Summe/2;

  // Wir geben die Ergebnisse aus:
  cout « "\n"       // Zeilenvorschub, sieht einfach besser aus
  « "Die Summe von " « Eingabe1 « " und " « Eingabe2
  « " ist " « Summe « ".\n";
  cout « "Das Produkt dieser Zahlen ist " « Produkt « ".\n";
  cout « "Ihr arithmetisches Mittel ist " « Mittelwert
  « " (abgerundet).\n";
}
```

Das Programm fängt ganz gewohnt an: Wir benutzen wieder die Definitionen aus <stream.h> und deklarieren die Hauptfunktion „main". Der Anweisungsteil beginnt dann auch sofort mit zwei Zeilen mit Variablendefinitionen. Wie Sie aus dem zuvor gesagten messerscharf schließen können, ist hier „int" jeweils der Name des Datentyps.

Eine Variable des Typs „int" kann eine ganze Zahl enthalten. Diese kann positiv oder negativ sein, darf aber nicht beliebig groß oder klein werden, nämlich so ungefähr im Bereich von -2000000000 bis +2000000000 (in Worten: plus/minus zwei Milliarden). Es gibt in C++ noch weitere Zahl-Datentypen, die z. B. noch größere oder auch gebrochene oder nur positive Zahlen umfassen. Ich möchte diese aber lieber in einem späteren Kapitel im Überblick einführen, denn für unsere ersten Programmierversuche reicht der Datentyp „int" vollkommen aus.

Die erste „int"-Zeile definiert also zwei Variablen eben diesen Typs. Ihre Namen - hier „Eingabe1" und „Eingabe2" - sind Bezeichner, die wir frei wählen können, einmal abgesehen davon, daß wir natürlich die im vorherigen Abschnitt genannten Regeln für Bezeichner beachten müssen und daß wir jeden Namen nur einmal verwenden können. Wir wählen hier die Namen so, daß man daran sofort erkennen kann, was die Variablen zu bedeuten haben, daß sie nämlich für Eingaben des Benutzers gedacht sind. Wenn das aus Längengründen nicht möglich ist - und das ist bei komplexeren Programmen der Normalfall - sollten Sie hinter die Definition einen Kommentar setzen, der die Bedeutung der Variablen erläutert.

☞ Wichtig:
In dem Moment, wo eine Variable definiert wird, hat sie noch keinen (bzw. einen zufälligen, undefinierten) Wert!

Die zweite „int"-Zeile definiert drei weitere Variablen, ebenfalls mit aussagekräftigen Namen. Es hat keinen besonderen Grund, daß wir die Definition hier auf zwei Zeilen aufgeteilt haben, genauso gut hätten wir auch alle fünf Variablen in einer Zeile definieren können. Auch müssen die Variablendefinitionen in C++ nicht immer am Anfang einer Funktion stehen: es reicht, wenn sie vor ihrer ersten Benutzung definiert werden.

Übrigens: „int" ist kein Bezeichner, sondern ein Wortsymbol wie „void". Und auch „void" bezeichnet einen Datentypen - allerdings den leeren Typen, der keinerlei Werte umfaßt und auch gar keine irgendwie besonderen Eigenschaften hat. Deshalb ist es auch nicht erlaubt, eine Variable des Typs „void" zu definieren. Der Typ „void" kann nur zu ganz bestimmten Zwecken eingesetzt werden, z. B. als Pseudo-Rückgabetyp einer Funktion, die gar kein Ergebnis liefert.

Nun folgen also die Anweisungen unseres Programms. Deren erste verstehen Sie ja bereits: Der Benutzer wird aufgefordert, zwei Zahlen einzutippen. Neu ist hingegen „cin". Das ist, genau wie „cout", ein Datenobjekt, nur steht es für die Standardeingabe statt -ausgabe. Der Operator „»", angewandt auf das Objekt „cin" und eine Variable, liest

einen Wert des passenden Typs - hier also Int-Zahlen - vom Standard-Eingabemedium und weist ihn der Variablen zu. Also wartet das Programm hier, bis der Benutzer zwei Zahlen eingegeben hat, und weist diese dann den beiden Variablen zu.

Natürlich kann nicht nur der Anwender (durch eine Texteingabe) einer Variablen einen Wert zuweisen - das kann das Programm auch selbst. Dazu dient in C++ der Operator „=".

Wie Sie sehen, steht linkes vom „=" immer eine Variable und rechts ein Ausdruck, der der Variablen zugewiesen werden soll. Ein Ausdruck besteht in C++ im wesentlichen aus Variablen, Konstanten und Operatoren. Dabei funktioniert alles, wie Sie es aus der Mathematik gewohnt sind: „+" addiert, „*" multipliziert, „/" dividiert, es gilt die Regel „Punktrechnung vor Strichrechnung", und Sie können bei Bedarf auch Klammern setzen, und zwar runde: „()". Zu erwähnen ist noch, daß der Divisionsoperator, wenn er auf ganze Zahlen angewendet wird, auch ein ganzzahliges Ergebnis liefert, indem er bei der Division auftretende Nachkommastellen einfach abschneidet. Außer den in unserem Programm benutzten Operatoren gibt es noch das „-" für die Subtraktion, das auch als Vorzeichen verwendet werden kann, und den Modulo-Operator „%", der den bei einer Division auftretenden Rest liefert, und noch jede Menge andere.

Nach den drei Zuweisungsoperationen haben also unsere drei Variablen geeignete Werte, die dann in drei „cout"-Zeilen ausgegeben werden. Hier haben wir mehrfach den Zeilenvorschub „\n" benutzt, um zu erreichen, daß jeder Wert in eine eigene Zeile ausgegeben wird. Wir hätten natürlich auch alle drei „cout"'s zu einem einzigen Ausdruck zusammenfassen können, aber so bleibt es wenigstens noch halbwegs übersichtlich.

1.4 Numerische Datentypen und Operatoren

1.4.1 Ganzzahlige Datentypen

Den wohl wichtigsten und meistgebrauchten numerischen Typen kennen Sie ja schon, nämlich „int". Dieser Datentyp ist, wie man so schön sagt, 32 Bit „breit", wobei ein Bit für das Vorzeichen gebraucht wird. Der Wertebereich geht damit von -2147483648 bis +2147483647 (daß der Bereich im negativen Bereich um 1 größer ist, kommt indirekt daher, daß es nicht sinnvoll ist, zwischen +0 und -0 zu unterscheiden. Die redundante -0 wird gewissermaßen zur Codierung einer zusätzlichen negativen Zahl benutzt).

Oft hat man es in Programmen aber auch mit Zahlen zu tun, die gar nicht negativ werden können, und dann wäre es doch Verschwendung, ein Bit für das Vorzeichen zu benutzen. Außerdem ist es doch auch viel schöner, wenn man den Umstand, daß man es mit positiven Zahlen zu tun hat, im Programmtext ausdrücken kann. Deshalb gibt es auch eine vorzeichenlose Variante von int, naheliegenderweise „unsigned int" genannt.

Einmal davon abgesehen, daß der Typname hier aus zwei Schlüsselwörtern besteht, deklariert man eine vorzeichenlose Variable genauso wie ein gewöhnliches „int", z. B. so:

```
unsigned int IrgendNeVariable;
```

- wobei man auch das „int" weglassen kann:

```
unsigned NochNeVariable;
```

Der Wertebereich solcher Variablen geht dann von 0 bis 4294967295.

Nicht immer - genaugenommen sogar recht selten - hat man es mit großen Zahlen zu tun. Oft kann man Speicherplatz sparen, wenn man schmalere Datentypen benutzt. Unter bestimmten Umständen spart das sogar Rechenzeit, da der normale 68000-Prozessor ja eigentlich nur ein 16-Bit-Prozessor ist. Also gibt es die Datentypen „short int" sowie „unsigned short int", die nur 16 Bit breit sind und einen entsprechend geringeren Wertebereich haben, den Sie der Tabelle auf der übernächsten Seite entnehmen können. Auch hier ist es Ihnen wie auch bei den folgenden Datentypen freigestellt, das „int" einfach wegzulassen.

Die genauen Datenbreiten sind leider in keinem Standard definiert, es gibt nur Mindestwerte. Deshalb gibt es auch C-Compiler, die allerdings langsam aussterben, mit nur 16 Bit breiten „int"-Typen. Die wirklich großen Datentypen heißen dann „long int" bzw. „unsigned long int". MaxonC++ kennt diese Datentypen natürlich auch, aber sie sind eben nur genauso lang wie die entsprechenden „int"-Typen. „int" und „long int" sind aber trotzdem formal zwei verschiedene Datentypen und nicht nur zwei verschiedene Schreibweisen für denselben Typen!

Dafür hat MaxonC++ eine kleine Besonderheit: 64 Bit breite Zahlen! Ursprünglich war nämlich vorgesehen, die „long"-Typen 64 Bit breit zu wählen, aber aus Kompatiblitätsgründen wurde dann davon abgesehen. Statt dessen gibt es jetzt die Datentypen „long long int" und „unsigned long long int", die ca. 19-stellige Zahlen aufnehmen können - ein nettes Feature, das man wohl nur selten brauchen wird, aber was man hat, das hat man. Es sei hier aber vor gedankenloser Benutzung dieser Datentypen gewarnt: Der Prozessor kann intern nur mit 32-Bit-Werten rechnen (und extern, wie oben angedeutet, sogar nur 16 Bit lesen und schreiben). Deshalb muß ein Programm für so ziemlich jede 64-Bit-Operation eine Bibliotheksfunktion aufrufen, entsprechend langsam geht das dann.

Übrigens: Man kann nicht nur das „int" am Ende optional weglassen. Es ist auch erlaubt, bei den vorzeichenbehafteten Datentypen „signed" davorzusetzen, etwa „signed int" (was nichts anderes als „int" ist) oder „signed long long".

Für Programmierer, die von Pascal oder ähnlichen Sprachen auf C umsteigen, ist es oft befremdlich, daß ein Zeichen in C nichts anderes als eine Zahl ist. Das ist im Prinzip natürlich vollkommen korrekt, denn der Computer kann nun einmal ausschließlich mit Zahlen arbeiten, so daß es intern keinen Unterschied zwischen einer 8 Bit breiten Zahl und einem Zeichen gibt. Deshalb kann man in C++ (Ästheten würden hier „leider" hinzufügen) den Datentypen „char" sowohl für Zahlen als auch für Zeichen verwenden - kommt ganz darauf an, wie man es betrachtet.

Folgerichtig gibt es auch Zeichen (oder sollte man besser „Ein-Byte-Zahlen" sagen?) mit und ohne Vorzeichen, also die Datentypen „signed char" und „unsigned char". Die Wertebereiche können Sie wieder der Tabelle am Ende dieses Abschnitts entnehmen.

Aber ist „char" nun identisch mit „signed char" oder „unsigned char"? Der ANSI-C-Standard läßt diese Frage offen, und damit hier gar keine Verwirrung entsteht, ist festgelegt, daß die drei Datentypen alle verschieden sind. In der Praxis ist „char" in MaxonC++ aber identisch mit „signed char".

So, und nun kommt endlich die versprochend Tabelle der Datentypen von MaxonC++:

Typ	von	bis
signed char	-128	127
char	-128	127
unsigned char	0	255
short	-32768	32767
unsigned short	0	65535
int	-2147483648	2147483647
unsigned int	0	4294967295
long	-2147483648	2147483647
unsigned long	0	4294967295
long long	-9223372036854775808	9223372036854775807
unsigned long long	0	18446744073709551615

1.4.2 Literale für ganze Zahlen und Zeichenketten

Ein „Literal" ist ganz allgemein eine vom Compiler akzeptierte Schreibweise für eine Konstante. In dem kleinen Programm

```
void main()
{ int i;
  i = 42;
}
```

ist „42“ eine Schreibweise für die bekannte und beliebte Zahl 42. Das klingt natürlich noch nicht sehr aufregend, aber es gibt in C noch viele andere mögliche Schreibweisen für ganze Zahlen.

Zunächst kann man im Literal angeben, welchen Typ die Zahl haben soll. Ein angehängtes „u“ oder „U“, auch „Suffix“ genannt, macht aus der Zahl einen vorzeichenlosen Wert, während das Suffix „l“ oder „L“ den Datentypen „long int“ erzwingt.
Beispiele:

```
42     ist int
42U    ist unsigned
42L    ist long
42UL   ist unsigned long
```

MaxonC++ kennt außerdem noch die Suffixe „LL“ und „ULL“ für die ganz breiten Datentypen:

```
42LL   ist long long
42ULL  ist unsigned long long
```

Außerdem gibt es in C noch verschiedene Zahlenbasen, nämlich Hexadezimal und Oktal. Hexadezimale Zahlen beginnen mit „0x“ oder „0X“ und erlaubte Ziffern sind ‘0’ bis ‘9’, ‘a’ bis ‘f’ bzw. ‘A’ bis ‘F’. Wie Sie sehen, ist bei Zahlendarstellungen die Groß-/Kleinschreibung generell nicht signifikant. Zum Beispiel hat „0xABCD“ den Wert 43981 und „0X2A“ ist 42.

Weniger gebräuchlich sind inzwischen die oktalen Zahlen, also die mit der Basis 8. Deshalb ist ihre Schreibweise wohl eher ein Relikt aus der Frühzeit von C: oktal sind nämlich alle Zahlen, die mit einer Null beginnen. „052“ ist also dezimal 42 und nicht etwa 52!

Immerhin ist man inzwischen so weit (seit ANSI-C), daß ‘8’ und ‘9’ keine oktalen Ziffern sind. In alten Zeiten war „08“ eine gültige Schreibweise für 8 - das gibt es jetzt nicht mehr. Jetzt gehen die Ziffern von ‘0’ bis ‘7’, wie sich das gehört.

Bekanntlich gibt es in C keinen wirklichen Unterschied zwischen Zahlen und Zeichen, weshalb in diesem Abschnitt auch char-Literale behandelt werden sollen. Für Zeichen verwendet man einfache Anführungszeichen, z. B.

```
char c, d;
c = 'X';
d = c+1;
```

Als Ergebnis erhält „d“ hier übrigens das Zeichen, dessen Code um 1 größer als der von ‘X’ ist, nämlich ‘Y’.

Genau wie bei konstanten Zeichenketten kann man auch hier die nicht-darstellbaren Steuerzeichen benutzen:

```
char c;
c = '\n';
```

weist „c" das Zeilentrennerzeichen zu, das übrigens auch LF (Linefeed) genannt wird und den Code 10 hat. Es werden sowohl bei Zeichen- als auch bei Zeichenkettenkonstanten folgende Steuer- und sonstige Zeichen unterstützt:

\n	Zeilentrenner (LF)
\t	Tabulator
\b	Backspace (ein Zeichen zurückgehen)
\r	Wagenrücklauf (Cursor an Zeilenanfang setzen)
\f	Seitenvorschub (Bildschirm löschen)
\a	Klingelzeichen
\\	das Gegenschrägstrich-Zeichen selbst
\?	Fragezeichen
\'	einfaches Anfügrungszeichen (z. B. als Charkonstante '\'')
\"	doppelte Anführungszeichen (z. B. in Stringkonstanten)

Das Fragezeichen kann man natürlich auch ohne vorgestellten Gegenschrägstrich in Strings verwenden. Es gibt in ANSI C allerdings die sog. Trigraph-Sequenzen, mit denen man den C-Standard auch auf Rechnern mit einem etwas seltsamen Zeichensatz mehr oder weniger benutzbar machen will. Dabei kann man jeweils zwei Fragezeichen, gefolgt von einem dritten Zeichen, als Ersatz für einige ASCII-Zeichen verwenden. Z. B. ist „??=" eine Ersatzschreibweise für „#", und „??)" kann man überall als Ersatz für die eckige Klammer „]" benutzen. Das braucht Sie auf Ihrem Amiga eigentlich nicht zu stören, denn Sie haben ja alle notwendigen Zeichen auf Ihrer Tastatur (auch wenn Sie evtl. manchmal etwas suchen müssen). Probleme gibt es höchstens dann, wenn Sie in einer Zeichenkette zwei Fragezeichen hintereinander verwenden und zufällig das falsche Zeichen dahinter setzen:

```
cout << "(Bahnhof??)";
```

gibt nicht etwa „(Bahnhof??)", sondern „(Bahnhof]" aus. Wenn Sie aber vor das zweite Fragezeichen einen Gegenschrägstrich setzen, können Sie diese Ersetzung unterdrükken, denn der Compiler ersetzt erst Trigraph-Sequenzen durch ihre Entsprechungen und erkennt dann erst Sonderzeichenfolgen in Strings.

Wenn Sie ein ganz besonderes Zeichen wollen, können Sie auch den entsprechenden ASCII-Code in Strings einsetzen. Sie haben die Wahl zwischen oktaler und hexadezimaler Angabe des Zeichencodes: Hinter den Backslach setzen Sie entweder bis zu drei oktale Ziffern oder ein „x", gefolgt von beliebig vielen Hexadezimal-Ziffern. Das Linefeed-

Zeichen „\n“ kann man also auch durch „\12“ oder „\x0a“ ersetzen, was natürlich die Portabilität des Programms unnötig einschränken würde. Allerdings ist es üblich (und auch legal), das Nullzeichen, das in C Strings abschließt, '\0' zu schreiben. Man könnte hier natürlich auch die Zahlenkonstante 0 verwenden, denn es gibt ja keinen Unterschied zwischen Zeichen und Zahlen, aber die erste Schreibweise ist anschaulicher, da klargemacht wird, daß hier wirklich das Zeichen 0 gemeint ist.

1.4.3 Ganzzahlige Operatoren und Berechnungen

1.4.3.1 Allgemeines

C und C++ bieten dem Programmierer eine reiche Auswahl an Operatoren. Die einfachsten arithmetischen Operatoren „+“, „-“, „*“ und „/“ sowie den Zuweisungsoperator „=“ haben Sie ja bereits kennengelernt. Hier gilt die aus der Mathematik bekennte Regel „Punktrechnung vor Strichrechnung“, d. h. 1+2*3 entspricht „1+(2*3)“. Wenn Sie hier die Auswertungsreihenfolge ändern wollen, können Sie entsprechend (runde) Klammern setzen. Außerdem ist zu erwähnen, daß die Bindung von links nach rechts erfolgt: „26731-47-11-42“ entspricht erwartungsgemäß „((26731-47)-11)-42“ und nicht etwa „26731-(47-(11-42))“. Das ist insofern erwähnenswert, als daß es in C++ durchaus auch Operatoren gibt, bei denen die Bindung von rechts erfolgt.

Damit wir nicht völlig die Übersicht verlieren, möchte ich die Operatoren in der Reihenfolge ihrer Bindung vorstellen, d. h. zuerst die, die am stärksten binden.

1.4.3.2 Postfix-Operatoren

Vorerst interessieren uns hier nur die nachgestellten Inkrement- und Dekrement-Operatoren „++“ und „--“. Sie bieten eine einfache und übersichtliche Möglichkeit, den Wert einer Variablen um 1 zu erhöhen bzw. zu erniedrigen.
Beispiel:

```
int i;
i = 42;
i++;
```

Anschließend hat „i“ den Wert 43; bei „--“ wäre dementsprechend 1 subtrahiert worden. Als Operanden benötigen „++“ und „--“ einen sogenannten L-Wert. Bisher sind Variablen die einzigen L-Werte, die wir bisher kennengelernt haben. Die Bezeichnung kommt übrigens daher, daß ein L-Wert insbesondere auf der linken Seite des Zuweisungsoperators „=“ stehen darf, so wie in „i=42“. Der Ausdruck „41++“ wäre also nicht erlaubt.

Jeder Operator hat in C++ ein Ergebnis. „++" und „--" liefern als Ergebnis den Wert, den der L-Wert vorher (!) gehabt hatte. Also ist folgende Codesequenz nicht unbedingt schön, aber möglich:

```
int i, j;
i = 42;
j = i--;
```

Anschließend ist „i" wieder 41, während „j" den alten Wert von „i", also 42, erhält.

Das Ergebnis ist allerdings kein L-Wert. Folgende naheliegende Anweisung zum Addieren von 2 ist also **nicht erlaubt**:

```
i++ ++;
```

Da diese beiden Postfixoperatoren also nur höchstens einmal angewendet werden können, ist es eigentlich müßig, ob die Bindung nun von links oder von rechts erfolgt. Wir werden später aber noch andere Postfixoperatoren kennenlernen, und deshalb sei jetzt schon der Vollständigkeit halber erwähnt, das die Auswertung von links erfolgt - was auch einsichtig ist, denn wenn da obiges Beispiel erlaubt wäre, würde es natürlich als „(i++)++" ausgewertet, während „i(++ ++)" überhaupt keinen Sinn ergäbe.

Am Rande sei noch bemerkt, daß der Ursprung der Bezeichnung „C++" im Dunkeln liegt. Die wahrscheinlichste Deutung ist, daß hier der Inkrementoperator auf die Programmiersprache „C" angewandt wurde, wobei allerdings der Operator ein wenig unglücklich gewählt wurde, denn wie Sie jetzt wissen, ist das Ergebnis von „C++" ja der alte Wert von „C". Wesentlich aussagekräftiger wäre deshalb der Name „++C", womit wir auch schon bei der nächsten Gruppe von Operatoren sind:

1.4.3.3 Unäre Operatoren

Die Operatoren „++" und „--" können einem L-Wert auch vorangestellt werden. Dann wird zum Operanden ebenfalls 1 addiert bzw. subtrahiert, aber hier wird als Ergebnis der NEUE Wert des Operanden geliefert.

Beispiel:

```
int i, j;
i = 42;
j = ++i;
```

Nun sind sowohl „i" als auch „j" 43. Erfahrungsgemäß ist es für C-Anfänger gar nicht so einfach, diese beiden Anwendungsarten derselben Operatoren auseinander zu halten. Am besten prägen Sie sich ein, daß bei „++var" zuerst addiert und dann der Variablenwert gelesen wird, während es bei „var++" entsprechend umgekehrt ist (für „--" analog).

Vier weitere unäre Operatoren können ebenfalls auf ganzen Zahlen angewendet werden:

\+ Das Vorzeichen „+" macht nichts mit dem dahinterstehenden Ausdruck und ist somit ziemlich überflüssig.

\- Das Minus kann ebenfalls als Vorzeichen benutzt werden und liefert den negierten Wert des Operanden. Bei vorzeichenlosen Datentypen ist das Ergebnis theoretisch undefiniert. MaxonC++ bildet hier das sog. 2-Komplement der vorzeichenlosen Zahl, aber darauf kann man sich nicht bei allen Compilern verlassen.

! Dieser Operator wird allgemein „nicht" (bzw. „not") genannt und bildet die logische Negation des Operanden: !x ist 1, wenn x = 0 gilt, und sonst 1. Die Operation ist nicht umkehrbar, denn !!x ist nicht identisch mit x.

~ Die Tilde steht in C für bitweise Negation: Jedes einzelne Bit wird umgedreht, d. h. aus 1 wird 0 und umgekehrt. Bei vorzeichenbehafteten Werten gilt ~0 = -1, ~42 = -43. Der Operator ist übrigens seine eigene Umkehrung: ~~x ergibt in jedem Fall wieder x.

Da diese Operatoren (im Gegensatz zu „++" und „--") den Wert des Operanden nicht verändern, muß dieser kein L-Wert sein. Die Bindung erfolgt natürlich von rechts: „-~!++x" ist „-(~(!(++x)))". Man beachte auch, daß „-1" in C kein Literal, sondern ein Ausdruck mit dem Operator „-" und dem Literal „1" ist.

Unäre Operatoren binden schwächer als Postfix-Operatoren. Also ist der Ausdruck

```
-i--
```

erlaubt und entspricht

```
-(i--) ,
```

was übrigens ein Beispiel für die berüchtigte Lesbarkeit von C ist.

Ein weiterer unärer Operator ist „sizeof". Er wertet den Operanden nicht aus, sondern gibt nur an, wieviel Speicher er belegt. Das Ergebnis hat laut ANSI den Datentypen „size_t", der in der Includedatei <stdlib.h> definiert ist. Unter MaxonC++ ist „size_t" identisch mit „unsigned int".

Es gibt zwei verscheidene Anwendungsarten von „sizeof":

\- Der Operand ist kein Ausdruck, sondern der Name eines Datentyps. Dann ist er in runde Klammern zu setzen, etwa so:

```
sizeof(long int)
```

Dieser Ausdruck ergibt in MaxonC++ übrigens 4, oder genaugenommen 4u, denn das Ergebnis ist ja „unsigned int". In C++ ist garantiert, daß „sizeof(char)" 1 liefert.

- Als Operand wird ein Ausdruck angegeben, der in Klammern stehen darf, aber nicht muß. Dann wird als Ergebnis der Speicherplatzbedarf des Typs des Operanden geliefert. Da der Typ eines Ausdrucks in C++ schon beim Compilieren festgestellt wird, wird der Ausdruck zur Laufzeit des Programms gar nicht ausgewertet.

Beispiel:

```
int i, j;
i = 42, j = sizeof i++;
```

„j" erhält als Wert den Speicherplatzbedarf des Typs von i++, der identisch mit dem von „i" ist, nämlich „int". In MaxonC++ ist „sizeof(int)" 4. Da der Operand „i++" aber nicht ausgewertet wird, wird der Wert von „i" auch nicht verändert - es gilt weiter „i=42".

Der „sizeof"-Operator wird vor allem für maschinennahe „low level"- Operationen benötigt. In C ist er außerdem wichtig, damit man bei der Funktion „malloc" weiß, wieviel Speicher angefordert werden soll, aber daß geht in C++ viel schöner und eleganter. Dazu später mehr.

1.4.3.4 Multiplikative Operatoren

So heißen die drei Operatoren, die gleichen Vorrang wie „*" haben, nämlich „/", „%" und natürlich „*" selbst.

Die Multiplikation „*" kennen Sie bereits. Zur Division „/" ist nur zu sagen, daß das Ergebnis bei ganzzahligen Operanden ebenfalls ganzzahlig ist und stets abgerundet wird. Ist einer der Operanden negativ, so werden die Beträge dividiert und das Ergebnis bekommt dann ein negatives Vorzeichen. Sind beide Operanden negativ, werden auch die Beträge dividiert, und das Ergebnis ist positiv. Das „Abrunden" bezieht sich hier also auf den Betrag des Ergebnisses.

Und dann wäre da noch der Operator „%". Er liefert den bei der ganzzahligen Division auftretenden Rest. 17/4 liefert also den Quotienten 4, 17%4 den Rest 1. Hier gelten die gleichen Vorzeichenregeln wie bei der Division, was mathematisch nicht gerade schön ist: Nach der mathematischen Definition ist „a modulo b" stets eine Zahl zwischen 0 und b-1, z. B. gilt

```
-17 modulo 4 = 3
(-17/4 = -4.25, abgerundet also -5; 4*(-5) ist -20,
der Rest ist dann (-17)-(-20) = 3)
```

In MaxonC++ liefert „-17 % 4“ dagegen das Ergebnis -1, denn der Quotient ist wegen Abrundung des Betrags -4, 4*(-4) ergibt -16, der Rest ist also (-17)-(-16) = -1.

Warum diese unschöne Definition in MaxonC++? Der Grund ist ganz einfach der, daß der Prozessor MC68000, der in den meisten Amiga steckt, so rechnet.

Da wir gerade dabei sind, den Prozessor schlechtzumachen: er kann noch nicht einmal richtig multiplizieren und dividieren! Die eingebauten Multiplikationsbefehle können lediglich zwei 16-Bit-Zahlen multiplizieren, wobei das Ergebnis dann immerhin ein 32-Bit-Wert ist; bei der Division kann er nur einen 32-Bit-Wert durch eine 16-Bit-Zahl teilen, wobei Ergebnis und Rest dann höchstens 16 Bit breit sein dürfen, sonst ist das Ergebnis undefiniert. Z. B. liefert folgendes Programmfragment

```
int i, j;
i = 2, j = 50000;
cout « i*j;
```

normalerweise ein vollkommen falsches Ergebnis, denn 50000 ist schon zu groß für eine vorzeichenbehaftete 16-Bit-Zahl (also ein „short int“).

Was tun? Entweder kaufen Sie sich nun umgehend einen 68030-Karte für Ihren Amiga, denn der kann vernünftig multiplizieren. Alternativ können Sie aber auch die Option „IntMult 32“ im Pull-Down-Menü „Optionen:Compiler“ aktivieren. Dann benutzt MaxonC++ nicht die eingebauten Multiplikations- und Divisionsoperationen des Prozessors, sondern ruft jeweils Bibliotheksroutinen auf, die richtig rechnen.

Das hat allerdings den Haken, daß diese Funktionsaufrufe um einiges länger dauern als die Prozessoroperationen. Also sollten Sie sich überlegen, ob Sie wirklich die genauen Bibliotheksfunktionen brauchen, denn oft hat man es ja mit relativ kleinen Zahlen zu tun, die die 16-Bit-Grenze nicht überschreiten.

Es sei noch angemerkt, daß MaxonC++, genau wie alle anderen namhaften C-Implementierungen, arithmetische Überläufe nicht abfängt. Sie können also unter Umständen unsinnige Ergebnisse erhalten, wenn Sie Ihre Datentypen nicht breit genug wählen. Das gehört leider zum Konzept von C++ (ist aber eher eine ungeliebte Erbschaft von C): man programmiert ohne Netz und doppelten Boden. Die Philosophie ist nun einmal, daß ein C-Programmierer weiß, was er will. Als Trost für die geringere Sicherheit bekommt man aber höhere Geschwindigkeit, denn das laufende Programm muß keine Zeit darauf verschwenden, irgendwelche Fehler abzufangen.

1.4.3.5 Addition und Subtraktion

Zu den Operatoren „+“ und „-“ gibt es nicht viel zu sagen: sie liefern die Summe bzw. die Differenz ihrer beiden Operanden. Die Bindung erfolgt naheliegenderweise von links.

1.4.3.6 Shift-Operatoren

Die beiden Operatoren „«" und „»" kennen Sie schon: Damit werden Ausgaben nach „cout" geschrieben bzw. Eingaben aus „cin" geholt. Aber das ist nicht ihre ursprüngliche Funktion: C++ bietet nämlich die Möglichkeit, Operatoren auf Klassen zu überladen, und in <stream.h> hat man nun eben zwei Operatoren auf den Klassen „ostream" bzw. „istream" überladen, um Objekte dieser Klassen - nämlich „cout" und „cin" - mit anderen Ausdrücken zu verknüpfen und so eine flexible Schreibweise für Ein- und Ausgaben zu erhalten. Daß man gerade die Operatoren „«" und „»" dafür gewählt hat, hat keine tiefsinnigere Bedeutung.

In Wirklichkeit haben „«" und „»" nämlich eine ganz andere Aufgabe: Sie shiften Bits nach links bzw. nach rechts.

Beispiel:

```
int i, j, k;
i = 13, j = i « 2;
cout « j;
```

Die Zahl 13 hat die Binärdarstellung 1101. Im Ausdruck „i « 2" werden alle Bits um zwei Positionen nach links geschoben und rechts wird mit Nullen aufgefüllt, was 110100 ergibt - und das ist dezimal 52 und somit die Ausgabe unseres Programmfragments. Ein n-facher Linksshift entspricht einer Multiplikation mit 2^n.

Der Rechtsshift-Operator „»" arbeitet analog, nur werden die Bits eben nach rechts verschoben, was einer Division durch 2^n entspricht. Bei vorzeichenbehafteten Datentypen bleibt das Vorzeichen erhalten. Übrigens binden auch diese Operatoren von links nach rechts.

Shift-Operationen braucht man vor allem für maschinennahe Bit-Manipulationen. Es ist ein beliebter Trick, Multiplikationen und Divisionen durch Verschiebungen zu ersetzen, denn mit einem Shift kann man erheblich Rechenzeit sparen. MaxonC++ optimiert aber schon selbstständig „8*i" zu „i«3", so daß Sie sich solche Bitfummeleien sparen können.

Wenn sie auch vollkommen unterschiedliche Bedeutungen haben, so haben die beiden bisher beschriebenen Bedeutungen von „«" doch den gleichen Auswertungsvorrang. Z. B. wird

```
cout « i « j;
```

ausgewertet wie

```
(cout « i) « j;
```

d. h. es wird erst „i“ und dann „j“ ausgegeben. Wenn Sie etwas anderes wollen, müssen Sie

```
cout « (i « j);
```

schreiben. Entsprechend ist

```
cout « i » j;
```

ein Fehler, denn hier werden zuerst „cout“ und „i“ verknüpft und und dann das Ergebnis und „j“ mit „»“ verknüpft. Da der Operator „»“ aber auf dem Typen von „cout“, nämlich der Class „ostream“, gar nicht definiert ist, meldet der Compiler hier einen Fehler. Hier müssen Sie also auf jeden Fall Klammern setzen.

Dieser scheinbar unpraktische Sachverhalt kommt daher, daß man in C++ zwar Operatoren überladen und umdefinieren kann, so wie es mit „«“ auf „ostream“ getan wird, aber nicht ihre Auswertungsreihenfolge verändern kann.

1.4.3.7 Vergleiche

Es gibt hier zwei Gruppen von Vergleichen mit unterschiedlichem Vorrang. Da wären zunächst die Vergleiche „kleiner“ (<), „kleiner oder gleich“ (<=), „größer“ (>) und „größer oder gleich“ (>=). Sie vergleichen zwei Zahlenwerte und liefern eine 1, wenn die Bedingung wahr ist, und sonst eine 0. Die Bindung geschieht von links nach rechts, aber Vorsicht:

```
0 < n < 1000
```

wird ausgewertet als

```
(0 < n) < 1000
```

Der Ausdruck in den Klammern liefert 0 oder 1, was stets kleiner als 1000 ist. Also sind obige Ausdrücke immer wahr! Ob eine Zahl wirklich zwischen 0 und 1000 liegt, müßte man mit

```
(0 < n) && (n < 1000)
```

abfragen - den Operator „&&“ lernen wir weiter unten kennen.

Dann gibt es die Vergleichsoperatoren == („gleich“) und != („ungleich“). Sie werden nach den anderen Vergleichsoperatoren ausgewertet:

```
a < b == c > d != 1
```

entspricht

```
((a < b) == (c > d)) != 1,
```

was logisch äquivalent zu

```
!((a < b) == (c > d))
```

ist (Sie erinnern sich: „!" war die logische Negation) und somit

```
(a < b) != (c > d)
```

entspricht. Das Ganze ist dann so etwas wie ein „ausschließliches Oder", denn der Ausdruck ist wahr, wenn einer der beiden Teilausdrücke - aber nicht beide - wahr ist. Und damit hätte ich auch schon elegant zu den diversen logischen Operatoren übergeleitet.

Achtung: C-Anfänger haben oft ihre Probleme mit dem Vergleichsoperator ==, denn viele andere Programmiersprachen verwenden hier das einfache „=". Fehler wie „n = 5" statt „n == 5" kommen anfangs oft vor und sind dann schwer zu finden. Aber ich kann Sie beruhigen: Man gewöhnt sich daran.

1.4.3.8 Bitweise Verknüpfungen

Neben dem bereits erwähnten unären bitweisen „Nicht"-Operator „~" bietet C++ noch die Bit-Verknüpfungen „&" (UND), „^" (AUSSCHLIESSLICHES ODER) und „|" (ODER). Von diesen hat „&" die höchste und „|" die niedrigste Auswertungspriorität, ansonsten erfolgt die Auswertung von links nach rechts. Die ganzzahligen Operanden werden Bit für Bit logisch verknüpft. Dabei werden alle Operanden ausgewertet, auch wenn das manchmal nicht nötig wäre: Bei Ausdrücken wie „0 & x" ist das Ergebnis ohnehin 0, so daß der Teilausdruck „x" eigentlich nicht weiter betrachtet werden müßte. Dies geschieht aber trotzdem, anders als bei den nun folgenden logischen Verknüpfungen:

1.4.3.9 Logische Operatoren

In C steht die Zahl 0 für „falsch" und jede andere Zahl für „wahr". Es gibt also keinen speziellen booleschen Datentypen wie in Pascal. Das ist oft ganz praktisch, denn statt „i != 0" kann man deshalb ganz einfach „i" schreiben, oder auch „!i" statt „i == 0".

Logische Aussagen können mit „Und" sowie „Oder" verknüpft werden. Die Operatoren dazu heißen „&&" bzw. „||" und liefern als Ergebnis stets 0 oder 1. Dabei wird die Auswertung ggf. angekürzt: ist bei „&&" der linke Operand 0, also logisch falsch, ist das Ergebnis sowieso 0 und der rechte Operand wird überhaupt nicht mehr betrachtet. Bei „||" entfällt entsprechend die Auswertung des rechten Operanden, wenn der linke bereits logisch wahr, also von 0 verschieden, ist.

Ein etwas obskures Beispiel:

```
int i, j;

j = i>=0 && i++;
```

Welchen Wert erhält hier „j“, und was geschieht mit „i“?

Wenn die linke Bedingung „i>=0“ falsch ist, ist das Ergebnis 0 und die Auswertung wird abgebrochen, der Wert von „i“ wird also nicht verändert. Falls „i“ aber doch positiv ist, muß die rechte Bedingung doch noch ausgewertet werden. Also wird dann „i“ um 1 erhöht und dann der alte Wert von „i“ betrachtet. Ist er logisch falsch, also 0, so erhält „j“ ebenfalls den Wert 0, andernfalls wird 1 zugewiesen.

Das Beispiel entspricht also:

```
if (i>=0)
{ j = (i != 0);
    i++;
}
else
  j = 0;
```

was natürlich wesentlich anschaulicher sind. Genaueres über „if“ und „else“ erfahren Sie in einem der folgenden Abschnitte.

Der Operator „&&“ wird vor „||“ zusammengafaßt. Ansonsten erfolgt die Auswertung logischer Verknüpfungen von links nach rechts.

1.4.3.10 Der bedingte Ausdruck

Wir haben bei den logischen Operatoren schon eine seltsame Möglichkeit kennengelernt, Fallunterscheidungen in Ausdrücken unterzubringen. Viel eleganter geht das mit dem Operator „?:“. Dies ist in C++ der einzige Operator, der gleich drei Operanden besitzt: Links vom Fragezeichen steht eine Bedingung. Ist diese Bedingung logisch wahr, so wird der Ausdruck zwischen „?“ und „:“ ausgewertet, andernfalls ist das Ergebnis der Ausdruck hinter dem Doppelpunkt.

Ein Beispiel:

```
sign = i>0 ? +1 : (i<0 ? -1 : 0);
```

Wie der Name schon andeutet, wird hier das Vorzeichen von „i“ berechnet. Gilt „i>0“, so wird nur „+1“ ausgewertet, was dann das Ergebnis ist, andernfalls wird ausschließlich

der Ausdruck hinter dem ersten „:“ ausgewertet, der ebenfalls ein bedingter Ausdruck ist. Dort wird dann noch auf „i<0“ getestet und das Ergebnis ist dann entweder -1 oder 0.

Im allgemeinen sollte man bedingte Ausdrücke vermeiden und statt dessen Fallunterscheidungen mit „if“ und „else“ benutzen:

```
if (i>0)
  sign = +1;
else
  if (i<0)
    sign = -1;
  else
    sign = 0;
```

Dies wird allgemein - zu Recht - als übersichtlicher angesehen.

Da wir gerade bei unübersichtlichen Ausdrücken sind: Die Bindung erfolgt hier von rechts, so daß der Ausdruck hinter dem „:“ ebenfalls ein bedingter Ausdruck sein kann, ohne daß man Klammern setzen muß!

Unser Beispiel könnte man (sollte man aber nicht) also auch so schreiben:

```
sign = i>0 ? +1 : i<0 ? -1 : 0;
```

Blicken Sie da noch auf Anhieb durch? Also, ich nicht! Also Pfoten weg von solchen Dingern!

Das Ganze wird noch chaotischer, wenn hinzter dem Fragezeichen beide Operatoren L-Werte vom gleichen Typ sind. Dann ist nämlich das Ergebnis des Ausdrucks ebenfalls ein L-Wert, dessen Wert man verändern kann:

```
int i, j;
i = 13, j = 14;

++(i>j ? i : j);
```

Hier wird der Operator „++“ entweder auf „i“ oder „j“ angewendet, anhängig von der Bedingung „i>j“. Es wird somit die größere Variable inkrementiert; in diesem Fall ist das „j“.

Wie Sie sehen, tragen bedingte Ausdrücke nicht gerade zur Lesbarkeit von Programmen bei. Es gibt aber durchaus Situationen, in denen normale Fallunterscheidungen mit „if“ und „else“ den Code unangemessen aufblähen würden, so daß man bedingte Ausdrücke manchmal nur schlecht vermeiden kann.

1.4.3.11 Zuweisungen

Den Zuweisungsoperator „=“ kennen Sie ja bereits. Der linke Operand muß ein L-Wert sein, der rechte ein Wert, der in den Typen des linken umgewandelt werden kann. Da wir bisher nur ganzzahlige Datentypen kennen, lassen sich Typen vorerst grundsätzlich ineinander umwandeln. Dabei kann es zu Überläufen kommen, wenn etwa ein zu gorßer Wert an einen schmalen Datentypen zugewiesen wird, was C aber größzügig ignoriert.

Dann gibt es noch die Zuweisungen mit Operator:

```
+=  -=  *=  %=  «=  »=  &=  |=  ^=
```

Jeder Ausdruck

```
A1 op= A2
```

ist eine Kurzschreibweise für

```
A1 = (A1) op (A2),
```

wobei „A1“ aber nur einmal ausgewertet wird.

„n += 1“ ist also identisch mit „++n“.

Man beachte, daß „i *= 2 + 3“ keineswegs „i = i * 2 + 3“, sondern „i = i* (2+3)“ bedeutet.

Jeder Operator hat in C ein Ergebnis, also auch eine Zuweisung. Ihr Ergebnis ist der Wert des linken Operanden nach der Zuweisung, was dann aber kein L-Wert ist.
So ist

```
a = b = c;
```

identisch mit

```
a = (b = c),
```

was eine beliebte Aukürzung für

```
b = c; a = b;
```

Die Bindung erfolgt hier offensichtlich von rechts nach links.

1.4.3.12 Der Komma-Operator

Die Benutzung des Kommas als Operator ist, ähnlich wie die bedingten Ausdrücke, ein C-Feature, daß manchmal einige Tipparbeit erspart, die Lesbarkeit von Programmen aber alles andere als verbessert. Trotzdem sei dieser Operator der Vollständigkeit halber erwähnt.

Es ist möglich, zwei oder mehr Ausdrücke mit „ , " aneinanderzuhängen. Sie werden der Reihe nach bewertet - es ist garantiert, daß die Auswertung von links nach rechts erfolgt, was z. B. bei „+" nicht versprochen wird - und das Ergebnis ist dann der Wert des letzten Ausdrucks.

Ein Beispiel:

```
int i, j;

j = (i=4, i++, i*i);
```

Hier wird zuerst der Variablen „i" der Wert 4 zugewiesen. Anschließend wird ihr Wert um 1 erhöht und zu guter Letzt das Quadrat von „i" berechnet. Dieser Wert, also 25, wird dann an „j" zugewiesen.

1.4.4 Fließkommatypen

1.4.4.1 Literale und Typen

Bisher haben wir uns nur mit ganzen Zahlen befaßt aber es gibt in C natürlich auch Fließkommazahlen, sogar drei Stück. Sie heißen „float", „double" und „long double". MaxonC++ benutzt nur zwei verschiedene Zahlenformate, so daß „double" und „long double" praktisch identisch sind. Formal gelten sie aber trotzdem als zwei verschiedene Datentypen.

Eine Fließkommakonstante besteht aus einem Vorkommateil, einem Komma, das aber in Wirklichkeit ein Punkt ist, einem Nachkommateil und optinal dem Exponentenzeichen „E" mit einem ganzzahligen Exponenten mit oder ohne Vorzeichen. Folgendes sind gültige Literale für Fließkomma- bzw. Gleitpunktzahlen:

```
17.4
26731.
.42
1.2e+3
1E10
0.05e-6
```

Wie Sie sehen, dürfen der Vorkomma- oder der Nachkommateil fehlen, aber natürlich nicht beide (welchen Wert hätte denn auch die Zahl „.“?) Wird ein Exponent angegeben, darf auch der Dezimalpunkt weggelassen werden. Für Anfänger sei noch erklärt, daß „1.2E+3“ die auf Computern übliche Schreibweise für „1.2*10^3“ ist und damit „1200.0“ entspricht.

Eine solche Fließkommakonstante hat aus unerfindlichen Gründen automatisch den Typ „double“. Durch Anhängen von „f“ oder „F“, z. B. „4.2f“, wird daraus eine „float“-Konstante. Entsprechend macht man sich mit dem Suffix „l“ oder „L“ eine „long double“-Konstante.

1.4.4.2 Interne Darstellung der Daten

Was sind überhaupt die Unterschiede zwischen den beiden (bzw. drei) Fließkommatypen? Und was sind überhaupt „Fließkommatypen“?

Eine Fließkomma- oder Gleitpunktzahl besteht aus Vorzeichen, Mantisse und Exponent. Ihr Wert ist dann

```
Vorzeichen * Mantisse * Basis^Exponent
```

Als „Basis“ wählt man naheliegenderweise die Zahl 2, die Mantisse wird stets auf ein bestimmtes Format „normiert“. Z. B. ist die Zahl 6 gerade 0.75 * 8, also 0.75 * 2^3. Die Mantisse ist hier also 0.75, der Exponent +3, und das Vorzeichen natürlich positiv. Jede Zahl läßt sich theoretisch in der Form 0.xxxxx * 2^y darstellen - praktisch stehen für die Mantisse aber nur endlich viele (Binär-)Ziffern zur Verfügung. Deshalb lassen sich Zahlen in der Regel nur als NÄHERUNG darstellen. Schon eine scheinbar so einfache Zahl wie 0.1 hat keine endliche Fließkommadarstellung, da die Mantisse ja binär und nicht dezimal dargestellt werden muß.

Ein „float“ ist vier Bytes, also 32 Bit, breit. Davon sind 24 Bits Mantisse, 1 Bit steht für das Vorzeichen und 7 Bits für den Exponenten. Das reicht für eine Genauigkeit von ca. 6 Stellen im Bereich von 10^-18 bis 10^+18. Überschreitet eine Rechnung diesen Bereich, tritt ein Überlauf auf, der aber keine Folgen hat, sondern „nur“ ein völlig falsches Ergebnis liefert. Alle Zahlen unter etwa 10^-18, oder „1E-18“, wie Computer zu sagen pflegen, betrachtet die Amiga-Arithmetik als Null, für die es eine Sonderdarstellung gibt (die Mantisse wird dabei nicht normiert, sondern einfach auf Null gesetzt).

„float“ ist zwar schon ganz nett, aber für viele Berechnungen reicht die Genauigkeit oder der Exponentenbereich einfach nicht aus. Deshalb gibt es „double“ bzw. „long double“: Diese Datentypen sind 8 Bytes bzw. 64 Bit breit. Sie setzen sich zusammen aus 52 Mantissenbits, 11 Bits für den Exponenten und natürlich wieder einem Vorzeichenbit.

Die Genauigkeit der Mantisse beträgt dann fast 16 Dezimalstellen („fast", weil es in der 16ten Ziffer zu Ungenauigkeiten kommen kann) und es sind dezimale Exponenten von +308 bis -308 möglich. Die größte intern darstellbare „double"-Zahl ist etwa 1.79769313486232e+308. Wenn Sie diese Zahl allerdings ausgeben wollen, kommt es dabei in den Bibliotheksfunktionen von MaxonC++ zu Überläufen, deshalb können nur maximal halb so große Zahlen mit „cout" ausgegeben werden. Der entsprechende Effekt tritt auch auf, wenn Sie sehr kleine Zahlen im Bereich von 1E-308 ausgeben wollen. Aber diese Einschränkungen sind wohl eher akademisch, denn in der Praxis sollte man es ohnehin vermeiden, bis an den äußersten Rand eines Wertebereichs zu gehen.

MaxonC++ benutzt für die Arithmetik die eingebauten Fließkommaroutinen des Amiga-Betriebssystems. Die „double"-Funktionen haben dabei die unangenehme Eigenart, daß bei einem Überlauf oder einer Division durch 0 eine Prozessor-Exception ausgelöst wird. Die integrierte Umgebung von MaxonC++ fängt solche Exceptions zwar ab, aber wenn ein solches Programm autonom läuft, hängt sich der Task auf. Also sollten Sie in der Tat vorsichtig mit „double"-Operationen umgehen.

1.4.4.2 Fließkommaoperationen

Alle artihmetischen Operatoren, die Sie von den ganzzahligen Typen her kennen, arbeiten, soweit das sinnvoll ist, auch auf Fließkommadaten. Der Modulo-Operator „%" ist nicht zulässig, denn bei einer Fließkommadivision tritt naturgemäß kein Rest auf. Ebenso sind die Operatoren „«" und „»" mit einem Fließkommaausdruck als linkem Operanden verboten, denn „double" und „float" haben ja intern ein etwas kompilziertes Format, und es würde keinen Sinn machen, irgendwelche Mantissenbits in den Exponenten hineinzuschieben. Auch die bitweisen Verknüpfungen „&", „|" und „^" sind aus demselben Grund für Fließkommaoperanden gesperrt. Folglich sind auch „%=", „«=", „»=", „&=", „|=" und „^=" nicht erlaubt.

Ein Fließkommaausdruck kann auch als „Bedingung" in einer bedingten Anweisung auftreten. Der Ausdruck ist dann „wahr", wenn er nicht 0.0 ist. Ebenso kann ein Fließkommaausdruck auch Operand der logischen Operatoren „||" und „&&" sein. Das Ergebnis der logischen Operatoren ist dann aber trotzdem „int".

1.4.5 Auswertung von Ausdrücken

Spätestens jetzt, wo wir zwei grundsätzlich verschiedene Gruppen von Datentypen (nämlich ganzzahlige und Fließkommatypen) kennen, kann ich Ihnen die genauen Auswertungsregeln nicht ersparen. Vor allem werden Sie bestimmt wissen wollen, was geschieht, wenn Operatoren auf Operanden unterschiedlicher Typen arbeiten.

Betrachten wir einmal einen Ausdruck wie „x + y". Hier gelten dieselben Regeln wie bei den meisten anderen binären Operatoren:

- Ist ein Operand „long double", so wird der andere in „long double" umgewandelt. Der Operator wird dann in „long double" ausgewertet und das ist auch der Typ des Ergebnisses; andernfalls:

- Ist ein Operand vom Typ „double", so wird der Operator entsprechend in „double" ausgewertet, sonst:

- Gilt dasselbe analog für „float".

Also wird bei „17.0 + 4", wo der linke Operand „double" ist, die 4 ebenfalls nach „double" konvertiert, und das Ergebnis ist „double".

Ist aber keiner der beiden Operanden ein Fließkommaausdruck, so gelten die Ganzzahl-Regeln:

Zuerst einmal werden Operanden der „char"- und „short"-Typen (gleich ob vorzeichenbehaftet oder vorzeichenlos) in „int" umgewandelt. Der Compiler lehnt es nun einmal ab, sich mit Zahlen zu befassen, die weniger als 32 Bit breit sind.

- Wenn nun einer der Operanden „unsigned long" ist, so wird der Ausdruck in „unsigned long" ausgewertet; wenn nicht:

- Ist ansonsten einer der Operanden „long", so erfolgt die Auswertung in „long", andernfalls:

- Falls ein Operand „unsigned int" ist, wird auch in eben diesem Datentyp gerechnet.

Wenn alle diese Bedingungen nicht zutreffen, wird der Ausdruck in „int" berechnet.

Ein beliebter Fehler ist, „1/2" zu schreiben. Wenn Sie sich die obigen Regeln einmal ansehen, werden Sie feststellen, daß hier in „int" gerechnet wird. Da aber bei der Integerdivision grundsätzlich das Ergebnis auch „int" ist und dementsprechend abgerundet wird, ist „1/2" nichts anderes als „0".

Folgende Schreibweisen würden hingegen zum gewünschten Ergebnis, nämlich dem „double"-Wert „0.5", führen:

```
1.0/2
1/2.0
1.0/2.0
```

Die hier beschriebenen Umwandlungsregeln gelten für die meisten binären Operatoren. Dazu gehören nicht die logischen Operatoren „||“ und „&&“, denn deren Ergebnis ist immer „int“ (genaugenommen 0 oder 1) und eine Umwandlung der Operanden erfolgt nicht, denn es kommt ja gar nicht auf deren Wert an, sondern nur darauf, ob ihr Wert Null ist oder nicht.

Bei den Vergleichsoperatoren werden zwar die Operanden genau wie oben beschrieben auf einen einheitlichen Typen gebracht, aber das Ergebnis ist auch hier stets ein logischer „int“-Wert.

Dabei kann es durchaus zu Überraschungen kommen, wenn vorzeichenlose und vorzeichenbehaftete Datentypen gleicher Größe verglichen werden:

```
int i;
unsigned int u;

i = -1;
u = +1;

cout « (i > u);
```

Verblüffenderweise ist das Ergebnis hier „1“, d. h. „-1“ ist größer als „+1“. Der Grund ist ganz einfach der, daß hier im vorzeichenlosen Bereich verglichen wird. Die „int“-Zahl „-1“ wird also un „unsigned“ umgewandelt (was ja eigentlich gar nicht geht) und entspricht dort „0xffffffff“, also der größten in „unsigned“ überhaupt darstellbaren Zahl. Grund für diese komische Umwandlung ist die sog. 2-Komplement-Darstellung für negative Zahlen, wie sie heute auf praktisch allen Rechnern angewandt wird.

Der Vollständigkeit halber sei noch erwähnt, daß bei den Zuweisungsoperatoren selbstverständlich die obigen Regeln nicht gelten. Bei der Zuweisung „=“ wird der rechte Operand in den Typen des linken umgewandelt. Dabei dürfen C-Compiler eine Warnung ausgeben, wenn ein Fließkommawert an eine ganzzahlige Variable zugewiesen wird.

Bei den zusammengesetzten Operatoren wie „+=“ oder „*=“ gilt dann wieder, daß „a op= b“ absolut identisch mit „a = a op b“ ist. So ist

```
int i;

i *= 0.5;
```

durchaus (bedingt) sinnvoll: es wird „i * 0.5“ wie oben beschrieben in „double“ ausgewertet, das Ergebnis wieder nach „int“ gewandelt und an „i“ zugewiesen.

Da aber so gut wie jede Typumwandlung Rechenzeit kostet (Ausnahme: Umwandlung zwischen Ganzzahltypen gleicher Breite), ist dies auch ein Beispiel für schlechte Programmierung, denn

```
i /= 2;
```

oder (bei manchen Compilern sogar noch besser)

```
i >>= 1;
```

würde das gleiche Ergebnis viel schneller liefern.

1.4.6 Explizite Typumwandlungen

Da wir gerade dabei sind, mit Datentypen zu jonglieren, möchte ich noch kurz ein Thema anreißen, das C berüchtigt gemacht hat, aber so schlimm nun auch wieder nicht ist: die expliziten Typkonvertierungen, auch „casting" genannt.

Sie erinnern sich wahrscheinlich noch an das Beispiel, bei dem wir den Compiler gezwungen haben, „1/2" als Fließkommaberechnung auszuwerten: es genügte, eine der beiden Konstanten als „double"-Literal zu schreiben, also beispielsweise „1.0/2". Aber was geschieht nun, wenn wir es mit Variablen statt Konstanten zu tun haben, etwa so:

```
int i, j;
double d;

i = 1;
j = 2;

d = i/j;
```

Der Ausdruck „i/j" wird streng nach den C-Typregeln erst als „int" ausgewertet, was also wieder „0" ergeben würde, und erst dann wird der Quotient nach „double" gewandelt, was offensichtlich nicht erwünscht ist.

Natürlich könnte man die C-Sprachdefinition ändern, etwa so, daß Ausdrücke, die an „double" zugewiesen werden, auch in „double" ausgewertet werden. Das erscheint auf kurze Sicht als eine sinnvolle und praxisgerechte Definition. Aber die Typregeln für Operatoren mit zwei Operanden sind schon kompiziert genug, und wenn man diese Regeln dann auch noch vom „gewünschten" Ergebnistyp abhängig macht, werden sie nur noch komplizierter, und dem Programmierer wird das Leben keinesfalls erleichtert.

Also zurück zu unserem Problem. Wir lösen es im Prinzip genau wie beim „1/2"-Beispiel: wir sagen dem Compiler, daß (wenigstens) einer der Operanden in ein „double" verwandelt wird. Und genau dazu dienen in C die „casts".

Die Sytnax ist zunächst denkbar einfach:
Ein Cast sieht aus wie „(typ) ausdruck", d. h. man schreibt erst den Ziel-Datentyp in runde Klammern und setzt dann den entsprechenden Teilausdruck dahinter.
Also sieht unsere Division folgendermaßen aus:

```
d = (double) i / j;
```

Ein Cast bindet so stark wie ein unärer Operator, aber von rechts nach links. Also ist

```
(double)i++ + j
```

identisch mit

```
( (double) (i++)) + j
```

Auch hier kann man natürlich mit Klammern eingreifen, z. B.

```
(int)(i+j)
```

In C++ gibt es eine alternative Schreibweise für Typwandlungen, die allgemein als lesbarer angesehen wird:

```
int(x)
```

Bei dieser Schreibweise wird also der Typname ohne Klammern geschrieben, während der umzuwandelnde Ausdruck geklammert wird. Bei einfachen Typnamen, wie sie bisher nur besprochen wurden, ist dies eine feine Sache. Wir werden aber bald Beschreibungen für abgeleitete Datentypen kennenlernen, und dann kommt man nicht mehr um die alte Cast-Form herum.

1.4.7 Initialisierungen

Oft hat man, wenn man eine Variable deklariert, schon einen passenden Anfangswert dafür zur Hand, was dann etwa so aussehen könnte:

```
int i, j;
i = 26, j = 731;
```

Weil diese Situation so oft vorkommt, gibt es in C die Möglichkeit der „Initialisierung" von Variablen. Das heißt ganz einfach, daß bei der Variablendeklaration ein Startwert gleich zugewiesen wird, etwa so:

```
int i = 26, j = 731;
```

Für Initialisierungen gelten in C die gleichen Regeln wie bei einer normalen Wertzuweisung. C++ hat einige Features auf Lager, wobei das nicht so ist, aber das werden wir noch beizeiten sehen.

Ein weiterer Unterschied zwischen C und C++: In C müssen in einem Block, also z. B. einer Funktion, immer zuerst die Variablendeklarationen und dann die Anweisungen stehen. In C++ darf man beides nach Belieben mischen:

```
void main()
{
  cout « "Bitte eine Zahl eingeben: ";  // Anweisung
  int i;        // Deklaration
  cin » i;      // Anweisung
  int j = i*i; // Deklaration
  cout « "Das Quadrat davon ist " « j;  // Anweisung
}
```

Der angenehme Effekt dabei ist, daß man Variablen erst dann deklarieren muß, wenn man einen sinnvollen Wert hat, um sie zu initialisieren. Das hilft, Fehler zu vermeiden, denn es ist im Beispiel nicht möglich, versehentlich „j" zu benutzen, bevor es initialisiert ist - ganz einfach deshalb, weil es da noch gar nicht bekannt ist.

1.5 Anweisungen

1.5.1 Bedingungen und Verzweigungen

Zu den wichtigsten Strukturelementen eines Programms gehören sicher Fallunterscheidungen. Im Prinzip gehören bedingte Ausdrücke auch dazu: In Abhängigkeit von einem logischen Ausdruck wird entschieden, welcher Teil eines Ausdrucks ausgewertet werden soll.

Die „if"-Anweisung dient zu demselben Zweck, aber auf höherer Ebene: Hier wird zwischen ganzen Anweisungen verzweigt. Ein zugegebenermaßen nicht sehr gehaltvolles Beispiel:

```
#include <stream.h>

void main()
{
  cout « "Bitte eine Zahl eingeben: ";
  int Ein, Aus;
  cin » Ein;

  if (Ein % 2 == 0)
  {
    cout « "Die Zahl ist gerade.\n";
    Aus = Ein/2;
    cout « "Die Hälfte von " « Ein « " ist " « Aus;
  }
  else
    cout « Ein « " ist ungerade.";
}
```

Eine Zahl ist bekanntlich gerade, wenn sie sich ohne Rest durch 2 teilen läßt, d. h. wenn „Zahl % 2" gerade Null ist. Dies ist die Bedingung, nach der im Beispiel verzweigt wird. Wie Sie hier sehen, muß die Bedingung hinter dem Wortsymbol „if" immer in runden Klammern stehen. Als Bedingung ist ein beliebiger numerischer Ausdruck (oder auch ein Zeigerausdruck, wie wir später kennenlernen werden) erlaubt. In C gibt es schließlich, wie bereits erwähnt, keine Trennung zwischen logischen und numerischen Ausdrücken.

Nur wenn die Bedingung wahr (d. h. nicht Null) ist, wird die folgende Anweisung hinter dem „if" ausgeführt. Es ist möglich, mehrere Anweisungen mit geschweiften Klammern zu einer einzigen zusammenzufassen, wie es in unserem Beispielprogramm geschehen ist.

Hinter den „if"-Zweig einer Fallunterscheidung kann - muß man aber nicht - noch einen „else"-Teil setzen, der ausgeführt wird, wenn die Bedingung eben nicht wahr ist. Er besteht aus dem Wortsymbol „else", gefolgt von einer Anweisung. Auch hier können natürlich mehrere Einzelanweisungen mit geschweiften Klammern zusammengefaßt werden.

Es gibt also genaugenommen zwei „if"-Anweisungen: Wenn ein „else"- Zweig angegeben wird, entscheidet die Bedingung darüber, welche der beiden Alternativen ausgeführt wird. Ohne „else" hängt von der Bedingung ab, ob der „if"-Zweig ausgeführt wird. In allen Fällen wird anschließend hinter der gesamten „if"-Anweisung weitergearbeitet.

Da eine „if"-Anweisung natürlich selbst eine Anweisung ist, kann man Fallunterscheidungen selbstverständlich auch ineinander verschachteln. Es gibt hier übrigens eine kleine „Zweideutigkeit":

```
if (Bedingung1)
if (Bedingung2)
  IfTeil;
else
  ElseTeil;
```

Zu welchem „if" gehört nun das „else"? Es wurde festgelegt, daß es zum zweiten „if" gehört. Also ist obige Anweisung identisch mit

```
if (Bedingung1)
{
  if (Bedingung2)
    IfTeil;
  else
    ElseTeil;
}
```

Durch entsprechende Klammerung kann man das „else" auch an das erste „if" binden:

```
if (Bedingung1)
{
  if (Bedingung2)
    IfTeil;
}
else
  ElseTeil;
```

Am Rande sei noch darauf hingewiesen, daß es dem Compiler absolut egal ist, wie und ob Sie Anweisungen einrücken. Der Programmierer tut dies ausschließlich für sich selbst und wer noch so alles den Quelltext lesen soll. Wenn man also schreiben würde:

```
if (Bedingung1)
  if (Bedingung2)
    IfTeil;
else
  ElseTeil;
```

wäre für den menschlichen Leser klar, was gemeint ist: das „else" soll zum ersten „if" gehören. Dem Compiler ist das vollkommen egal, für ihn gelten auch nur hier seine geradezu bürokratischen Vorschriften, nach denen das „else" nun mal zum zweiten „if" gehört, und damit basta. Seien Sie also beim Verschachteln von Verzweigungen

vorsichtig, im Zweifelsfall sollten Sie immer Klammern setzen, damit unmißverständlich klar wird, was wozu gehört. Das ist also ganz analog zum empfohlenen Setzen zusätzlicher runder Klammern in komplizierten Ausdrücken.

1.5.2 Anweisungen, Deklarationen und Blöcke

Leider wird es wieder einmal Zeit, daß wir uns in Formalitäten ergehen (man könnte fast glauben, das Programmieren sei von Beamten erfunden worden...). Zunächst soll noch einmal präzisiert werden, was eine „Anweisung" ist. Wir kennen nämlich inzwischen vier verschiedene Arten davon:

1.) Ausdrucks-Anweisungen: Sie bestehen aus einem Ausdruck, der ausgewertet wird, gefolgt von einem Semikolon. Ein Ausdruck ist im wesentlichen alles das, was in 1.4 beschrieben wurde, nämlich Variablen und Konstanten, die durch Operatoren verknüpft werden. Auch ein trivialer Ausdruck wie „42" ist eine Anweisung, bei der aber nichts geschieht. MaxonC^{++} erzeugt nicht einmal Code für so etwas. Aber es gibt ja durchaus Operatoren, die Auswirkungen auf ihre Operanden haben oder sonst eine Operation einschließen, etwa der Zuweisungsoperator „=" oder die Ein- und Ausgabeoperatoren „«" und „»".

2.) Die „if"-Anweisung: evtl. mit „else"-Teil. Das Ganze Gebilde gilt dann nachher als eine einzelne Anweisung. Im folgenden werden wir noch viele ähnliche Strukturanweisungen betrachten, insbesondere Schleifenanweisungen.

3.) Eine Deklaration gilt in C++ ebenfalls als Anweisung. In C ist das anders: Hier müssen Deklarationen und Anweisungen streng voneinander getrennt werden, nämlich dergestalt, daß in einem „Block" immer zuerst alle Deklarationen und dann die Anweisungen stehen müssen.

4.) Eine Block-Anweisung: Bisher haben wir sie lässig damit abgetan, daß mehrere Anweisungen mit geschweiften Klammern „{ ... }" zusammengefaßt werden. Das ist aber bei weitem nicht alles, und darum soll dies im folgenden näher unter die Lupe genommen werden.

Zunächst einmal ist der Anweisungsteil einer Funktion, z. B. „main", denn andere Funktionen haben wir bis hier noch nicht gehabt, nichts anderes als ein „Block", und formal ist das eine Folge von Anweisungen (die in C++ ja, wie schon wiederholt gesagt, auch Deklarationen einschließen).

Vor allem aber hat ein Block einen eigenen „Gültigkeitsbereich", oder auf neudeutsch „Scope". Das bedeutet: Namen, die innerhalb eines Blocks deklariert werden, insbesondere Variablen - sind außerhalb gar nicht bekannt.

Ein Beispiel aus der Praxis:

```
#include <stream.h>

 void main()
{ // Anfang des Haupt-Blocks der Funktion "main"

  cout « "Wollen Sie in i)nt oder d)ouble rechnen? ";
  char i;
  cin » i;

  if (i == 'i')
  { // Hier beginnt ein Block
    cout « "Ihre Eingabe war: " « i « "\n";
    int i, j, erg;
    cout « "Bitte zwei ganze Zahlen: ";
    cin » i » j;
    erg = i/j;
    cout « "Quotient: " « erg;
  } // Das ist das Ende eines Blocks

  else
    if (i == 'd')
    { // Noch ein Block
      cout « "Danke für Ihre Eingabe " « i « ".\n";
      double x, y, erg;
      cout « "Bitte zwei Zahlen: ";
      cin » x » y;
      erg = x/y;
      cout « "Quotient: " « erg;
    } // Ende dieses Blocks
    else
    {
      cout « "Das war eine falsche Eingabe.";
    }

  cout «  "\nProgrammende - Variable \"i\
          " hat immer noch den Wert " « i;
  // Die einzige Variable, die an dieser Stelle bekannt ist

} // Funktionsende
```

Bei diesem kleinen Programm hat der Benutzer die Wahl, ob er ganzzahlig oder in Fließkomma dividieren will. Dazu wird zuerst ein Zeichen abgefragt, das dann in der „char"-Variablen „i" abgelegt wird. Nun folgt eine Verzweigung, deren „if"-Teil ein eigener Block ist.

Zunächst wird hier die „char"-Variable „i" ausgegeben, nur um zu zeigen, daß Bezeichner aus höheren Blöcken durchaus in tiefergelegenen (nicht „tiefergelegten", Manta-Witze sind schließlich total out) Blöcken bekannt sind und benutzt werden dürfen. Aber nun kommt etwas Überraschendes: In der folgenden Deklarationszeile wird unter anderem eine neue „i"-Variable definiert, diesmal sogar als „int"!

Das gehört zum Konzept der „Gültigkeitsbereiche": Bezeichner dürfen in verschiedenen Bereichen durchaus für ganz verschiedene Zwecke definiert und benutzt werden. In diesem Fall überdeckt das neue „i" das alte, so daß wir von jetzt an nur noch das neue „i" aus dem inneren Block benutzen können. Außerdem werden zusätzlich noch die Variablen „j" und „erg" definiert.

Der innere Block endet bei den geschweiften Klammern, und wir befinden uns nun wieder auf der Ebene des äußeren Blocks. Das bedeutet vor allem, daß die „innen" deklarierten Variablen „i", „j" und „erg" jetzt nicht mehr bekannt sind und nicht mehr benutzt werden können. Genaugenommen existieren diese Variablen überhaupt nicht mehr, ihr Speicherplatz könnte also prinzipiell wiederverwendet werden. Dafür hat „i" jetzt wieder die alte Bedeutung, nämlich als „char"-Variable mit der Benutzereingabe.

Nun ist da noch der „else"-Teil, ebenfalls mit einem eigenen Scope. Hier werden die Variablen „x", „y" und „erg" dazudefiniert. Das zuvor definierte „erg" war ja „lokal" zu einem inneren Block und deshalb auf Ebene der Funktion überhaupt nicht bekannt. Also ist es in diesem neuen Block erst recht nicht bekannt und kann getrost neu definiert werden, diesmal als „double".

Und dann hat auch noch der „else"-Teil der zweiten „if"-Anweisung einen eigenen Block. Weil dieser nur eine einzige Anweisung enthält und keinerlei neue Bezeichner definiert werden, wäre es überhaupt nicht nötig gewesen, hier einen neuen Block zu definieren.

Die letzte „cout"-Zeile gehört wieder zum obersten Block der Funktion. Die einzige Variable, die an dieser Stelle bekannt ist, ist „i". Alle Variablen aus tiefer liegenden Blöcken sind hier nicht mehr definiert. Nebenbei bemerkt wird hier auch demonstriert, wie man doppelte Anführungszeichen in einen String einschließt, indem man einen Backslash davorstellt.

Als Anweisung hinter „if" und „else" darf keine einzelne Deklaration stehen, z. B.

```
if (Bedingung)
  int i; // Error
else
  int j; // Error
```

Das ist ja auch klar, denn was soll obige Anweisung denn auch wohl bedeuten? Sollen „i" und „j" nur in Abhängigkeit von einer Bedingung existieren, oder wie oder was oder warum auch nicht? Mit einem Block ist das natürlich egal, etwa

```
if (Bedingung)
  { int i; }
else
  { int j; }
```

Hier sind „i" und „j" nur in ihren kleinen Blöcken bekannt, so daß die Gültigkeit wieder offensichtlich ist.

Übrigens: In C muß hinter jeder Anweisung ein Semikolon stehen. Die einzige Ausnahme ist ein Block, hier darf man das „ ; " weglassen. Manchmal, z. B. zwischen „if" und „else", muß man es sogar unterschlagen, denn ein überflüssiges Semikolon gilt in der C-Syntax als Leeranweisung, so daß bei

```
if (Beding)
  { Block; };       // letzteres Semikolon überflüssig!
else
  usw;
```

gleich zwei Anweisungen zwischen „if" und „else" stehen, was ja nicht zulässig ist.

1.5.3 Schleifen

1.5.3.1 Die „while"-Anweisung

Zu den unverzichtbaren Elementen der Programmierung gehören Schleifenstrukturen. Dabei wird allgemein eine Folge von Anweisungen wiederholt ausgeführt, bis eine wie auch immer geartete Endbedingung erfüllt ist.

Die elementarste Schleifenform ist in C die „while"-Schleife. Sie hat die Form

```
while (Bedingung) Anweisung;
```

Die „Anweisung" wird solange wiederholt, wie die „Bedingung" wahr (ungleich Null) ist. Möglicherweise ist die Bedingung von Anfang an nicht erfüllt - dann wird der Schleifenrumpf überhaupt nicht ausgeführt.

Ein Beispiel:

```
#include <stream.h>

main()
{
  int i = 1, sum = 0, max = 100;
  while (i <= max)
  { sum += i;
    i++;
  }
  cout « "Ergebnis: " « sum;
}
```

Die Variable „i" wird so lange von 0 hochgezählt, bis ihr Wert 100 überschreitet. Dabei werden die Werte jeweils in „sum" aufaddiert. Als Ergebnis wird dann die Summe der ganzen Zahlen von 1 bis 100 ausgegeben, nämlich 5050.

Echte C-Freaks hätten übrigens die Schleife abgekürzt:

```
while (i <= max)
  sum += i++;
```

Mathematiker wissen natürlich, daß man das Ergebnis auch direkt als

```
sum = max * (max+1) / 2;
```

berechnen könnte.

1.5.3.2 Die „for"-Schleife

Die Zählschleife tritt so häufig auf, daß es dafür eine verkürzte Schreibweise gibt: die „for"-Schleife. Sie hat die Form

```
for (Anweisung1; Bedingung; Ausdruck)
  Anweisung2;
```

Die erste Anweisung wird einmal zu Anfang ausgeführt. Dann wird die Schleife so lange ausgeführt, wie die Bedingung erfüllt ist, wobei der „Ausdruck" nach jedem Schleifendurchlauf ausgewertet wird.

Sie entspricht also

```
Anweisung1;
  while (Bedingung)
  {
  Anweisung2;
  Ausdruck;
  }
```

Dementsprechend können wir unser kleines Beispiel umschreiben:

```
#include <stream.h>

main()
{
  int i, sum = 0, max = 100;

  for (i=1; i <= max; i++)
    sum += i;

  cout « "Ergebnis: " « sum;
}
```

Im Gegensatz z. B. zu Pascal ist „for" in C ein sehr allgemeines Sprachkonstrukt. Man kann wegen der Verwandtschaft mit der „while"-Schleife weit mehr damit machen, als nur eine Variable rauf- oder runterzuzählen. Die Einsatzmöglichkeiten von „for" sind genauso vielfältig wie die von „while".

Bei „for" sind alle drei Ausdrücke bzw. Anweisungen in den Klammern optional. So ist „for (;;) Anweisung;" identisch mit einer endlosen Schleife (eine fehlende Schleifenbedingung wird als „falsch" betrachtet), und „for(;Bedingung;) Anweisung" entspricht vollkommen der „while"-Schleife.

In C++ darf man eine Schleifenzählvariable auch innerhalb der „for"-Startanweisung deklarieren:

```
for (int i=0; i != 10; —i)
  { BlaBla; }
```

Die Variable gehört dann aber zum Scope, in dem die „for"-Schleife liegt, und nicht zum Gültigkeitsbereich des Schleifenrumpfs. Deshalb dürfen Sie bei einer nachfolgenden Schleife „i" zwar wieder benutzen, aber nicht noch einmal deklarieren.

1.5.3.3 Die „do"-Schleife

Die dritte Schleifenkonstruktion von „C" ist die „do"-Anweisung. Bei „while" und „for" wird die Schleifenbedingung schon vor dem ersten Schleifendurchlauf getestet, und wenn sie von Anfang an „nicht erfüllt" ist, wird die Schleife keinmal durchlaufen. Bei „do" dagegen wird die Bedingung erst am Schleifenende und somit nach der ersten Ausführung des Schleifenrumpfs geprüft. Deshalb wird eine „do"-Schleife immer mindestens einmal ausgeführt.

Die „do"-Anweisung hat die Form:

```
do
  Anweisung;
while (Bedingung);
```

Zwischen „do" und „while" darf auch hier nur eine einzige Anweisung (bzw. ein Block) stehen. Bei der „repeat"-Anweisung in Pascal, die in etwa dem „do-while" von C entspricht, bilden ja „repeat" und „until" eine Klammerung um beliebig viele Anweisungen.

Auch die „do"-Schleife wird durchlaufen, so lange die Bedingung erfüllt ist (die „repeat"-Anweisung in Pascal wird ausgeführt, so lange sie **nicht erfüllt** ist).

Ein typisches Beispiel für „do" sind Wiederholungen von Benutzerfunktionen, etwa so:

```
char c1;
do
{
  DatenDrucken; // Irgendwelches Zeugs auf den Drucker ausgeben
  cout « "Soll ich die Daten noch einmal ausdrucken? (j/n) ";
  cin » c1;
}
while (c1 == 'j');
```

Im Gegensatz zur „while"-Schleife darf die Variable „c1", die in der Schleifenbedingung benutzt wird, vor dem ersten Schleifendurchlauf undefiniert sein, denn die Bedingung wird ja erst nach der ersten Ausführung des Schleifeninhalts getestet.

1.5.4 Sprünge

1.5.4.1 Sinn und Unsinn von „goto“

Schleifenstrukturen sind die wichtigsten Elemente strukturierter Programmierung. Es gibt aber Gelegenheiten, bei denen man es nur schwer umgehen kann, wie in alten FORTRAN- oder BASIC-Zeiten mit der „goto“-Anweisung quer durch ein Programm zu springen.

Das Ziel eines „goto“-Sprungs wird in C durch ein „Label“, das durch einen Bezeichner repräsentiert wird, bestimmt. Eine „goto“-Anweisung hat dann die Form

```
goto Label;
```

Die Sprungmarke kann man in der Form

```
Label: IrgendEineAnweisung;
```

vor eine beliebige Anweisung stellen. Dabei bilden dann die Marke und die nachfolgende Anweisung zusammen formal eine einzige Anweisung.

Solche Sprünge ermöglichen dann wirklich ganz scheußliche Programmabläufe. Ein Beispiel:

```
void main()
{
  int i, j=0, s;

  if (EgalWas == IrgendWas)
  {
    i = 0;
    goto Dahin;
  }

  for (i=10; i <= 20; ++i)
  {
    s = i*i;
    Dahin: j += s;
  }
}
```

Die ganze schöne Struktur der „for“-Schleife geht hier vor die Hunde: Wenn die obskure Bedingung „EgalWas == IrgendWas“ erfüllt ist, wird „i“ mit 0 initialisiert und ohne jede Gnade in die „for“-Schleife eingesprungen. Dabei wird auch die Schleifenanfangs-

anweisung übergangen, so daß jetzt nicht von 10, sondern von 0 aus gezählt wird. Dann benutzen wir zu allem Überfluß auch noch die Variable „s", die gar keinen definierten Wert besitzt, so daß „j" hier einen wahrscheinlich unsinnigen Wert erhält - ein typischer Bug, wie er durch unüberlegte „goto"'s oft verursacht wird.

Jeder Bezeichner hat bekanntlich einen Gültigkeitsbereich. Die Sprungmarke „Dahin" wird im obigen Programm aber scheinbar in einem Block definiert und in einem ganz anderen benutzt. Das scheint aber wirklich nur so, denn die Labels einer Funktion haben einen eigenen Scope. Sie kollidieren nicht mit anderen Bezeichnern der Funktion - man könnte also die Verwirrung komplett machen und denselben Namen für eine Variable und ein Label wählen - und sind in der gesamten Funktion gleichermaßen bekannt. Diese Verletzung der gewohnten Gültigkeitsregeln stellen einen Grund mehr da, auf Sprünge möglichst zu verzichten.

Immerhin wird bei „goto" aber ganz scharf darauf geachtet, daß keine Initialisierung übersprungen wird:

```
goto Dahin;
for (i=10; i <= 20; ++i)
{ int s = i*i;
  Dahin: j += s;  // Error: Jump past initialisation of "s"
}
```

Hier wird bei der Deklaration des Labels ein Fehler wie „Jump past initialisation of object 'i'" gemeldet, d. h. der Compiler stört sich daran, daß hinter die Initialisierung einer Variablen gesprungen wird. Das kommt daher, daß eine Initialisierung keine normale Wertzuweisung ist: Jedes Datenobjekt kann genau einmal, bei seiner Deklaration, initialisiert werden. Nun wird aber mit dem „goto" die Initialisierung von „i" übersprungen. Sinn und Zweck einer Initalisierung ist aber, daß Variablen von Anfang an einen „sinnvollen" Wert haben, deshalb wurde es generell verboten, auf irgendeine Art eine Initialisierung zu übergehen. Alternativ könnte man vielleicht festlegen, daß der Compiler solche Deklarationen irgendwie „herauszieht", so daß die Variable auf jeden Fall noch vor dem „goto" initialisiert wird. Der dazu nötige Aufwand stünde aber in keinerlei Relation zum Nutzen, und so hat man es beim schlichten Verbot solcher Strukturen belassen.

Es ist aber sinnvollerweise erlaubt, Initialisierungen von Variablen, die beim Sprungziel sowieso nicht mehr bekannt sind, zu überspringen:

```
for (Dieses; und; jenes)
{ i = IrgendEinWert;
  if (i > 1000)
    goto NotAusstieg;
  long TotalEgal = i*i;
}
NotAusstieg:
```

Die Initialisierung von „TotalEgal" darf getrost übersprungen werden, da diese Variable bei der Sprungmarke „NotAusstieg" nicht bekannt ist und folglich „total egal" ist, ob sie nun einen sinnvollen Wert hat oder nicht.

Abschließend ist zu sagen, daß man Sprünge nur verwenden sollte, wenn der normale Programmablauf gestört wird und es deshalb nötig ist, aus einer Struktur auszuspringen.

1.5.4.2 „break" und „continue"

Im letzten Beispiel wurde ein „goto" benutzt, um eine Schleife zu verlassen. Das geht in C aber auch eleganter: mit der „break"- Anweisung.

Ihre Syntax ist denkbar einfach: „break;".

Diese Anweisung beendet sofort die nächsthöhere „while"-, „for"-, „do"- oder „switch"-Anweisung (letztere wird weiter unten behandelt). Die Programmausführung wird dann hinter der jeweiligen Strukturanweisung fortgesetzt, so daß „break" einem „goto" zu einem imaginären Label hinter der Schleife entspricht.

Unser letztes Beispiel läßt sich eleganter (jedenfalls unter Vermeidung eines „goto") so schreiben:

```
for (Dieses; und; jenes)
{ i = IrgendEinWert;
  if (i > 1000)
    break;
  long TotalEgal = i*i;
}
```

Es gibt tatsächlich Situationen, bei denen man weder am Anfang noch am Ende einer Schleife feststellen kann, ob das Ende der Schleifenausführung erreicht sein soll, und diese Entscheidung vielleicht sogar an MEHREREN Stellen im Schleifenrumpf getroffen werden muß. Dann bietet sich folgende Konstruktion an:

```
for(;;)  // eigentlich eine Endlosschleife
{ EinPaarAnweisungen;
  if (IrgendWas)
    break;
  NochEinigeAnweisungen;
  if (WasWeissIch)
    { break };
  UndNochWas;
  // usw..
}
```

Wie Sie beim zweiten „if" erkennen, kann „break" durchaus aus mehreren Blöcken auf einmal ausspringen. Es bezieht sich eben, wie gesagt, auf die nächsthöhere Schleifen- oder „switch"-Anweisung.

Mit „break" kann man die meisten „goto" auf halbwegs strukturierte Weise ersetzen. Und es gibt noch eine weitere Anweisung, mit der man Sprünge vermeiden kann: „continue".

Auch diese Anweisung hat eine simple Syntax, nämlich „continue;". Sie bezieht sich auf die nächsthöhere Schleifenanweisung und setzt die Programmausführung am Anfang des Schleifenrumpfs fort, überspringt also den gesamten Rest des Rumpfs. Bei

```
for(Start; Bedingung; Ausdruck)
{
  Anweisung1;

  continue;

  Anweisung2;

  HierKoennteEinLabelStehen:
}
```

wird nach dem „continue" der „Ausdruck" ausgewertet, also z. B. die Schleifenzählvariable inkrementiert, die „Bedingung" geprüft und dementsprechend die Schleife fortgesetzt oder auch nicht. Ergo entspricht „continue" hier einem „goto HierKoennteEinLabelStehen".

Bei „while" sieht das Ganze ähnlich aus:

```
while (Bedingung)
{
  Anweisung1;

  continue;

  Anweisung2;

  // Ende des Schleifenrumpfs
}
```

„continue" übergeht „Anweisung2" und geht direkt ans Ende des Schleifenrumpfs bzw. zu der Stelle, an der die „Bedingung" getestet wird.

Auch bei „do“ ist die Bedeutung von „continue“ einleuchtend:

```
do
{
  Anweisung1;

  continue;

  Anweisung2;

  // Ende des Schleifenrumpfs
} while (Bedingung);
```

Auch hier wird zu der Stelle gesprungen, an der die Schleifenbedingung ausgewertet wird.

Mit „break“ und „continue“ dürfte es möglich sein, die meisten „goto“-Anweisungen zu ersetzen. Leider gibt es kein „sauberes“ Sprachkonstrukt, das die zweit- oder dritthöhere Schleife verläßt, so daß man dann doch auf ein „goto“ zurückgreifen muß.

1.5.5 Vielfachverzweigungen

Bisweilen muß man in Abhängigkeit von einem Wert verzweigen, und das nicht, wie bei einem einfachen „if“, zwischen zwei, sondern zwischen etlichen Programmstellen. Die „switch“-Anweisung stellt eine bequeme und schnelle Abkürzung für Strukturen der Art

```
if (val == Wert1)
  { ... }
else if (val == Wert2)
  { ... }
else if (val == Wert3)
  { ... }
else if (val == Wert4)
  { ... }
else ...
```

dar. Dabei muß „val“ allerdings ein ganzzahliger Ausdruck sein, und bei „Wert1“, „Wert2“ usw. muß es sich um konstante Ausdrücke handeln, die also bereits vom Compiler und nicht erst zur Laufzeit des Programms ausgewertet werden können.

Eine „switch"-Anweisung hat die Form

```
switch (val)
  Anweisung;
```

wobei man als „Anweisung" im allgemeinen wohl einen ganzen Block wählen wird. In diese Anweisung setzt man Sprungmarken der Form

```
case Wert:
```

Beim „switch" wird dann in Abhängigkeit von „val" zum zugehörigen „case" verzweigt.

Ein Beispiel:

```
cout « "Daten L)aden, S)peichern, D)rucken oder P)rogrammende:
";
char c;
cin » c;

switch (c)
{
  case 'l': case 'L':
    Laden;
    break;
  case 's': case 'S':
    Speichern;
    break;
  case 'd': case 'D':
    Drucken;
    break;
  case 'p': case 'P':
    cout « "Sind Sie wirklich sicher? "; cin » c;
    if (c != 'j' && c != 'J')
      break;
    ProgrammEnde;
      break;
  default:
    cout « "Falsche Eingabe!\n";
    break;
}
```

Der Benutzer wird hier aufgefordert, mit Eingabe eines Zeichens aus einem kleinen Menü auszuwählen. Als erste Alternative wird bei Eingabe von 'l' oder auch 'L' der Menüpunkt „Laden" ausgeführt. Das anschließende „break" ist unbedingt nötig, um aus dem „switch" auszusteigen, denn sonst würde ganz einfach in der nächsten Zeile

weitergemacht, also mit der Programmfunktion „Speichern". Hier wurde die Programmiersprache wohl nicht unbedingt sinnvoll definiert, denn es dürfte so gut wie nie vorkommen, daß man nach einer „switch"-Verzweigung ohne „break" gleich mit der nächsten Alternative weitermachen will, während man andererseits leicht ein solches „break" vergißt und sich dann wundert, was das Programm denn macht. Aber mit dieser Unzulänglichkeit von C wird man wohl leben können und müssen.

Jedenfalls ist das Ganze so gedacht, daß die „case"-Angaben lediglich Labels sind, zu denen beim „switch" wie mit einem „goto" verzweigt wird. Also darf man auch - wie oben geschehen - beliebig viele „case" vor eine einzige Anweisung setzen. Andererseits darf jeder Wert nur in einem „case" auftreten, z. B. könnte man nicht zweimal „case 'p'" verwenden - denn wohin sollte dann auch verzweigt werden?

Das Label „default" darf ebenfalls in jedem „switch" höchstens einmal auftreten. Hierhin wird verzweigt, wenn der fragliche Wert bei keinem „case" angegeben wird. Wenn kein „default:" angegeben wird, wird in diesem Fall die „switch"-Anweisung ganz übergangen. Übrigens muß „default" keineswegs das letzte Label sein.

Das „break" hinter der letzten Anweisung eines „switch" ist eigentlich überflüssig, denn am Ende wird die „switch"-Anweisung ja sowieso verlassen. Es ist aber empfehlenswert, es hier trotzdem zu setzen, denn oft hängt man an ein „switch" noch nachträglich Alternativen an, und dann vergißt man leicht das trennende „break" - Folgen: siehe oben.

Bei „switch" gelten im Zusammenhang mit Initialisierungen genau dieselben strikten Regeln wie bei „goto":

```
switch(i)
{ case 0:
    int j = 1;
  break;
  case 1:
    j = 26731;
    int k = j+1;
}
```

Hier beschwert sich der Compiler, daß bei der Verzweigung nach „case 1" die Initialisierung von „j" übersprungen wird. Allenfalls in der letzten Alternative darf, wie hier bei „k", eine Initialisierung stattfinden, denn die kann ja nicht mehr übersprungen werden, jedenfalls nicht innerhalb ihres Blocks. Deshalb bietet es sich an, einem „case" einen eigenen Block zu spendieren, um Ärger mit Initialisierungen lokaler Variablen zu umgehen:

```
switch(i)
{ case 0:
  { int j = 1;
    break;
  } // Hier ist der Block nötig
  case 1:
    j = 26731;
    int k = j+1;  // Das darf so bleiben
}
```

1.5.6 Die „return"-Anweisung

Das wären dann auch schon (fast) alle Anweisungstypen, die es in C++ gibt. Der Vollständigkeit halber soll hier noch die „return"-Anweisung erwähnt werden, über die Sie genaueres im Kapitel 1.6.3 erfahren. Sie beendet die Ausführung einer Funktion und kann optional auch einen Wert zurückgeben. Aber damit gehört sie auch schon in das folgende Kapitel:

1.6 Funktionen

1.6.1 „main" allein genügt nicht

Neben strukturierten Anweisungen - also etwa übersichtlicher Schleifenstrukturen statt wilder Sprünge - ist die Abstraktion durch Funktionsdefinition eine zentrale Idee der strukturierten Programmierung. Die Idee ist, weitgehend in sich abgeschlossene Programmteile zu einer Funktion zusammenzufassen und diese am besten auch noch so allgemein zu formulieren, daß sie möglichst allgemein zu benutzen ist.

Bisher hatten wir es immer mit nur einer Funktion zu tun, nämlich „main". Das ist insofern eine besondere Funktion, als daß der Linker bei der endgültigen „Montage" des Programms darauf besteht, daß es eine solche Funktion gibt, und dann dafür sorgt, daß die Funktion „main" beim Programmstart ausgeführt wird. Für den Compiler gibt es dagegen keinen formalen Unterschied zwischen der Funktion „main" und einer beliebigen anderen.

Wir können also z. B. eine Funktion „hello" definieren, zunächst einmal ohne Rückgabewert (also „void"):

```
#include <stream.h>

void hello()
{ cout « "Hello, World!";
}
```

Die leeren Klammern „()", die wir schon bei der Funktion „main" stets definiert haben, sind die „Parameterliste" der Funktion „hello". Da diese Liste offensichtlich leer ist, hat die Funktion keine Parameter.

Eine Funktion wird mit Namen und Parameterliste aufgerufen, z. B. so:

```
#include <stream.h>

void hello()
{ cout « "Hello, World!";
}

void main()
{
  hello();      // Funktion "hello" wird ausgeführt
  cout « "\n"; // Zeilenvorschub
  hello();      // Und noch einmal "hello"
}
```

Ungeheuer wichtig: Auch wenn die Funktion gar keine Parameter hat, muß man in C (im Gegensatz z. B. zu Pascal) beim Aufruf eine leere Parameterliste anhängen. Der Funktionsname allein bezeichnet nur die Adresse einer Funktion. Der Postfix-Operator „()" (in C gilt eine Parameterliste tatsächlich als Operator) angewendet auf den Namen einer Funktion ruft eben diese auf.

Ein Funktionsaufruf ist übrigens ein Ausdruck und hat deshalb auch einen Datentyp, nämlich jeweils den Ergebnistyp der Funktion. In unserem Fall ist das der Typ „void", so daß der Funktionsaufruf keinen wirklichen Wert hat und man mit dem Ergebnis auch nicht so sonderlich viel machen kann.

1.6.2 Parameter und Argumente

Jetzt sind Sie bestimmt schon neugierig, was es genau mit diesen Klammern „()" auf sich hat, denn wenn von einer **leeren** Parameterliste die Rede war, kann man bestimmt auch etwas da rein schreiben... Und genau davon soll jetzt die Rede sein.

Ein „Parameter" ist ein Wert, der beim Aufruf an eine Funktion übergeben wird. Praktisch geschieht das, indem eine Variable der Funktion „von außen" initialisiert wird. Diese Parametervariablen werden innerhalb der Parameterliste angegeben, z. B. so:

```
void f(int i, int j)
```

Aus historischen Gründen (sprich: wieder eine lästige Erbschaft von veralteten C-Standards) kann man hier **nicht** wie bei einer gewöhnlichen Variablendefinition

```
void f(int i, j) // Achtung, FALSCH!
```

schreiben. Aber die Anzahl der Parameter ist natürlich beliebig, und es können auch unterschiedliche Datentypen benutzt werden, z. B.

```
void NochEineFunktion(int i, char c, double d);
```

Jedenfalls hat die Funktion „f" nun zwei Variablen „i" und „j", die beim Aufruf initialisiert werden. Die folgende Funktion berechnet nach dem euklidischen Algorithmus den größten gemeinsamen Teiler seiner Parameter:

```
void ggt(int i, int j)
{
  cout « "Der größte gemeinsame Teiler von " « i « " und "
  « j « " ist: ";

  while (i) // Kurzschreibweise für "while (i != 0)"
  { int hilf = j % i;
    j = i;
    i = hilf;
  }
  cout « j;
}
```

Wie Sie sehen, kann man Parameter genau wie gewöhnliche Variablen benutzen und ihnen sogar einen anderen Wert zuweisen.

Beim Aufruf einer Funktion sind dann geeignete Werte für die Parameter anzugeben, etwa

```
void main()
{ int a, b;
  cout « "Bitte zwei Zahlen eingeben: "; cin » a » b;
  ggt(a,b);
}
```

In C werden normalerweise nur Werte als Parameter übergeben. Die Variablen „a" und „b" des Hauptprogramms behalten deshalb ihre Werte, auch wenn in der Funktion „ggt" den Parametervariablen neue Werte zugewiesen werden.

An dieser Stelle sind dringend einige Anmerkungen zur Sprechweise nötig: Man unterscheidet nämlich zwischen „Parametern" und „Argumenten". In unserem Beispiel sind „i" und „j" die Parameter der Funktion „ggt", während „a" und „b", also die Werte, mit denen die Parameter initialisiert werden, „Argumente" genannt werden. Dementsprechend ist

```
(int i, int j)
```

eine Parameterliste und

```
(a, b)
```

eine Argumentliste. In Pascal ist es dagegen üblich, von „formalen" und „aktuellen" Parametern zu sprechen, also auch von „formalen Parameterlisten" (statt „Parameterliste") bzw. „aktuellen Parameterlisten" (statt „Argumentliste").

Bei der Parameterübergabe gelten dieselben Typregeln wie bei Initialisierungen (das sind bisher noch dieselben wie bei einer gewöhnlichen Wertzuweisung). Also werden auch Datentypen von Argumenten in die der Parameter umgewandelt. Beim Aufruf

```
ggt(17.0, 4.2)
```

würden folglich die Integerwerte „17" und „4" an die Parameter zugewiesen, wobei C-Compiler wieder eine Warnung ausgeben dürfen, was MaxonC++ aber unterläßt.

1.6.3 Rückgabe von Werten

Wie schon zuvor angedeutet, muß der Ergebnistyp einer Funktion keineswegs unbedingt „void" sein. In unserem Beispiel mit dem größten gemeinsamen Teiler (ggt) wäre es doch ausgesprochen praktisch, wenn die Funktion das Ergebnis nicht nur einfach auf den Bildschirm ausgeben, sondern es an das aufrufende Programm zurückgeben würde, damit man es im weiteren Programmablauf verwenden kann. Das geht im Prinzip ganz einfach:

```
int ggt(int i, int j)
{
  while (i)
  { int hilf = j % i;
    j = i;
    i = hilf;
  }
  return j;
}
```

Zunächst einmal wird als Ergebnistyp der Funktion „int" angegeben, es wird also eine Zahl zurückgegeben. Unser Algorithmus läuft dann genau wie bei der ersten Version, nur daß wir keine Ausgaben machen. Am Ende unserer Funktion steht dann eine „return"-Anweisung. Hier wird die Funktion beendet und das Ergebnis, also der Wert von „j", an das aufrufende Programm zurückgegeben.

Dort können wir jetzt den Funktionsaufruf in Ausdrücken und Anweisungen benutzen, etwa so:

```
void main()
{ int a, b;
  cout « "Bitte zwei Zahlen eingeben: "; cin » a » b;
  int erg = ggt(a, b);
  cout » "Der ggt der Eingabe ist: " » erg
}
```

...oder auch so:

```
void main()
{ int a, b;
  cout « "Bitte zwei Zahlen eingeben: "; cin » a » b;
  cout » "Der ggt der Eingabe ist: " » ggt(a, b);
}
```

Da ein Funktionaufruf ein Ausdruck ist, kann er z. B. auch Argument einer anderen Funktion sein:

```
int Eingabe(int min, int max)
{ int input;

  for(;;)
  {
    cout « "Bitte eine Zahl zwischen " « min « " und " « max
    « "eingeben: ";
    cin » input;

    if (input >= min && input <= max)
      return input;

    cout « "Falsche Eingabe! \n";
  }
}

int ggt(int i, int j);       // wie oben

void main()
{
  cout « ggt(Eingabe(1,10000), Eingabe(1,10000));
}
```

In diesem Beispiel werden diverse Funktionsaufrufe ineinander verschachtelt. Zunächst wird im Hauptprogramm der Operator „«" ausgewertet, dessen Operanden (bzw. Argumente) das Objekt „cout" und der nachfolgende Ausdruck sind. Jener Ausdruck ist ein Aufruf der Funktion „ggt", wobei als Argumente jeweils die Funktion „Eingabe" aufgerufen wird. Diese Funktion erwartet eine Eingabe des Benutzers und prüft dabei, ob sie im erwarteten Bereich liegt.

Es stellt für das Programm also kein Problem dar, die Auswertung eines Ausdrucks zu unterbrechen, eine Funktion auszuführen und dann den Ausdruck zu Ende zu berechnen.

Übrigens ist obiges Beispiel ziemlich gewagt programmiert, denn der C++-Sprachstandard sagt nichts darüber aus, in welcher Reihenfolge die Argumente einer Funktion ausgewertet werden, ob hier also zuerst der linke oder der rechte Aufruf von „Eingabe" erfolgt. Bei der Berechnung des größten gemeinsamen Teilers spielt die Reihenfolge der Parameter aber keine Rolle, so daß hier unterschiedliche Implementierungen kein wesentlich anderes Ergebnis liefern würden. MaxonC++ wertet die Argumente von links nach rechts aus, aber um zu anderen Implementierungen kompatibel zu bleiben, sollten Sie sich nie darauf verlassen.

Zum Abschluß noch einige Anmerkungen zur „return"-Anweisung:

- Eine Funktion kann beliebig viele „return"-Anweisungen haben. Die Ausführung einer Funktion wird durch die „return"-Anweisung sofort abgebrochen und der nachfolgende Wert zurückgegeben.

- Auch in einer Funktion mit „void"-Ergebnis kann „return" benutzt werden. In diesem Fall wird natürlich kein Ergebniswert angegeben, man schreibt also ganz einfach „return;".

- Viele Programmierer setzen das Ergebnis einer „return"-Anweisung in runde Klammern, etwa „return (2*j);", wodurch das Ganze wie ein Funktionsaufruf aussieht. Das darf man machen (es ist erlaubt, in und um Ausdrücke überflüssige Klammern zu setzen), aber es hat keinerlei Bedeutung. Auch in diesem Handbuch wird auf diese Klammern verzichtet, denn „return" ist nun einmal keine Funktion, sondern eine eigenständige Art von Anweisung.

- Der Ausdruck hinter „return" wird nach den üblichen Regeln für Initialisierungen in den Ergebnistyp der Funktion umgewandelt.

Es ist weit verbreitet (und leider auch erlaubt), bei der Deklaration einer Funktion überhaupt keinen Ergebnistyp anzugeben, z. B.

```
Anonym()
{ // irgendwas...
  return;
  // nochwas ...
  return 42;
}
```

Wenn eine solche Funktion in einem Ausdruck aufgerufen wird, wird davon ausgegangen, daß ihr Ergebnis „int" ist. Innerhalb einer Funktion ohne Rückgabetyp darf „return" wahlweise mit oder ohne Ausdruck aufgerufen werden, während sonst die Schreibweise ohne nachfolgenden Ausdruck nur in „void"-Funktionen und die mit Ausdruck ausschließlich in Funktionen mit einen anderen Ergebnistyp als „void" benutzt werden darf.

Deklarationen von Funktionen ohne Ergebnistyp sind schon wieder eine Erbschaft von C.

In diesem Zusammenhang sei noch einmal deutlich gesagt, daß C nicht, wie z. B. Pascal, zwischen „Functions" mit Ergebnis und „Procedures" ohne Ergebnis unterscheidet. Eine Funktion mit einem Ergebnis kann ganz normal als Anweisung aufgerufen werden, wobei ihr Resultat dann ignoriert wird:

```
int f()
{ cout « "f wurde aufgerufen.\n";
  return 26731;
}

void main()
{ int i = f();  // Ergebnis von "f" wird benutzt
  f();          // Ergebnis von "f" wird ignoriert
}
```

1.6.4 Funktionen und Gültigkeitsbereiche

1.6.4.1 Deklarationen auf Dateiebene

Jede Funktion hat ihren eigenen Gültigkeitsbereich. Das heißt, daß eine Funktion nicht auf Variablen oder sonstige Bezeichner einer anderen Funktion zugreifen kann - ein Bezeichner, der innerhalb einer Funktion definiert wird, ist außerhalb nicht sichtbar. Also kann man natürlich denselben Namen in verschiedenen Funktionen definieren - diese Bezeichner haben dann für den Compiler absolut nichts miteinander zu tun.

Wenn Funktionen also gemeinsame Daten benötigen, muß die aufrufende Funktion der aufgerufenen die gewünschten Daten als Parameter übergeben. Dies ist oft nervig und manchmal gar nicht praktikabel, weshalb es noch eine weitere Möglichkeit gibt: Deklarationen auf Dateiebene. Ein Beispiel:

```
#include <stream.h>

int zahl1, zahl2, ergebnis;

void ggt()
{ int i = zahl1, j = zahl2;
  while (i)
  { int hilf = j % i;
    j = i;
    i = hilf;
  }
  ergebnis = j;
}

void main()
{ cout « "Bitte zwei Zahlen eingeben: ";
  cin » zahl1 » zahl2;
  ggt();
  cout « "ggT: " « ergebnis;
}
```

Die Variablen „zahl1", „zahl2" und „ergebnis" werden auf Dateiebene (man sagt auch „global") definiert. Ihr Gültigkeitsbereich liegt damit noch über dem der Funktionen. Sie sind in allen Funktionen sichtbar, können dort aber natürlich auch überdefiniert werden.

Das kleine Programm simuliert Parameterübergaben im BASIC-Stil: Die Parameter werden in Variablen abgelegt, woraus das „Unterprogramm" sie übernimmt und das Ergebnis berechnet, das dann in einer globalen Variablen abgelegt wird. Damit ist das Programm wieder einmal ein Beispiel dafür, wie man nicht programmieren sollte. Es gibt aber tatsächlich Situationen, in denen man globale Variablen benutzen muß. Dann sollte man aber auf jeden Fall einen Kommentar an den Anfang der betreffenden Funktionen setzen, der auf solche „versteckte" Parameter hinweist.

Es ist nicht nötig, die globalen Variablen vor den Funktionen zu definieren. Funktions- und Variablendeklarationen dürfen in C beliebig gemischt werden. Da man aber jede Variable deklarieren muß, bevor man sie benutzen kann, ist es in der Regel am sinnvollsten, erst alle Variablen und dann die Funktionen zu deklarieren.

Übrigens werden auch die Objekte „cin" und „cout" in der Includedatei „stream.h" als globale Variablen deklariert, das erklärt dann auch, warum sie in allen Funktionen ohne weiteres zur Verfügung stehen.

1.6.4.2 Rekursive Funktionen

Eine Variable, die innerhalb einer Funktion definiert wird, wird bei jedem Aufuf der Funktion auf den „Stack" erzeugt und nach Beendigung der Funktion wieder gelöscht. Das hat dann gewichtige Konsequenzen, wenn eine Funktion sich selbst aufruft (man nennt sie dann „rekursive Funktion").

Ein fast schon klassisches Beispiel für Rekursion: Die Fakultät einer Zahl n wird oft deklariert als Produkt aller Zahlen von 1 bis n, z. B.

```
5! = 1*2*3*4*5 = 120
```

Es gibt aber eine noch viel schönere Definition: Man setzt 0! = 1 und für alle anderen Zahlen gilt:

```
n! = n * (n-1)!
```

Nach dieser Definition ist dann 5! = 5 * 4! = 5 * 4 * 3! und so weiter. Und genau nach dieser Definition berechnet die folgende Funktion die Fakultät einer Zahl:

```
int fak(int n)
{ if (n == 0)
  return 1;
else
  return n * fak(n-1);
}
```

Es sollte Sie absolut nicht stören, daß die Funktion sich selbst aufruft, das tut sie nämlich nicht endlos oft (was man versehentlich auch öfters programmiert und dann zu einem Absturz führt, weil der Stack überläuft), sondern nur, bis die Variable „n" auf 0 heruntergezählt ist. Die Parametervariable „n" wird bei jedem einzelnen Aufruf neu erzeugt und anschließend wieder gelöscht, so daß die Rekursion nicht den Wert von „n" auf der aktuellen Ebene verändern kann. Genauso würde für weitere lokale Variablen, die man in „fak" deklarieren könnte, bei jeder Rekursion ein neuer Platz auf dem Stack eingerichtet. Damit gibt es keinen Unterschied zwischen einer Rekursion und einem gewöhnlichen Funktionsaufruf: Die lokalen Variablen einer Funktion werden in beiden Fällen garantiert nicht verändert, es sei denn, man wünscht das ausdrücklich - wie das geht, steht im Kapitel 2.

Prinzipiell kann man jede Schleife durch eine Rekursion ersetzen, so wie wir im letzten Beispiel eine simple Zählschleife zum Multiplizieren der ersten „n" Zahlen durch eine elegante rekursive Funktion ersetzt haben. Sie sollten es mit Rekursionen allerdings nicht übertreiben: Zum einen hat der Stack, auf dem bei jedem Funktionsaufruf Variablen und andere Daten abgelegt werden, nur eine begrenzte Größe, und wenn der überläuft, kommt der allseits beliebte Guru. Zum anderen benötigt ein Funktionsaufruf immer einiges an Zeit (Sie erinnern sich: Variablen müssen erzeugt und nachher wieder gelöscht werden, und auch sonst ist einiges internes Stack-Management nötig), so daß eine Schleifenstruktur oft weniger schön, aber stets schneller ist.

1.6.4.3 Deklarationen und Definitionen

Die Namen von Funktionen sind naheliegenderweise auf Dateiebene deklariert. Und spätestens jetzt ist es an der Zeit, einmal auf die Unterschiede zwischen einer „Deklaration" und einer „Definition" zu kommen.

Es ist nämlich durchaus möglich, eine Funktion zu deklarieren, ohne sie zu definieren. Betrachten wir einmal die beiden folgenden Funktionen:

```
int gerade (int);

int ungerade (int n)
{ if (n<0)
    return ungerade (-n);
  else
    return !gerade(n);
}

int gerade (int n)
{ if (n == 0)
    return 1
  else
    return ungerade(n-1);
}
```

Die beiden Funktionen stellen auf denkbar umständliche Art und Weise fest, ob eine Zahl gerade oder ungerade ist, nämlich rekursiv nach folgenden Regeln:

1. Eine negative Zahl ist ungerade, wenn die zugehörige positive Zahl ungerade ist.

2. Eine positive Zahl ist ungerade, wenn sie nicht gerade ist.

3. Null ist gerade.

4. Eine Zahl ist gerade, wenn die nächstkleinere Zahl ungerade ist.

Umständlicher geht es wirklich nicht, aber das steht hier nicht zur Debatte. Wesentlich ist vielmehr, daß die beiden Funktionen „ungerade" und „gerade" sich gegenseitig aufrufen. In C++ muß man aber jede Funktion deklarieren, bevor man sie benutzt (in C ist das anders, siehe auch den „Anachronismen"-Abschnitt). Deshalb wird zuerst die Funktion „gerade" nur deklariert: Ergebnistyp, Name und Parameterliste werden angegeben und das Ganze mit einem Semikolon beendet. Nun weiss der Compiler, was er mit dem Bezeichner „gerade" anzufangen hat: die Funktion „gerade" ist bekannt und darf deshalb aufgerufen werden. Man nennt eine solche Funktionsdeklaration ohne nachfolgende Definition auch „Prototyp" der Funktion.

Man beachte, daß hier in der Deklaration der Parameter den Typen „int", aber keinen Namen hat. Man darf sie durchaus wie gewohnt benennen, aber diese Bezeichner haben dann keinerlei Bedeutung. Es ist sogar erlaubt, eine Funktion beliebig oft zu deklarieren und jedesmal andere Bezeichner für die Parameter zu wählen. Es kommt lediglich darauf an, daß die Datentypen und die Reihenfolge der Parameter gleich bleiben.

Nach der Deklaration und gleichzeitiger Definition der Funktion „ungerade" folgt im obigen Beispiel die Definition, also der Funktionsrumpf, der bisher nur deklarierten

Funktion „gerade". Hier muß wieder die komplette Parameterliste angegeben werden (bei „forward"-Deklarationen in Pascal ist das ja anders). Es ist zu empfehlen, den Parametern hier Namen zu geben, denn sonst kann man sie in der Funktion ja nicht benutzen. Prinzipiell ist es aber erlaubt, namenlose Parameter zu deklarieren. Das kann z. B. sinnvoll sein, wenn man einen Parameter derzeit noch nicht benötigt, aber plant, die Funktion später so zu erweitern, daß ein zusätzlicher Parameter nötig wird. Dann kann man gleich in der Funktionsdeklaration durch einen namenlosen Parameter deutlich machen, daß eben dieser Parameter vorerst noch keine Bedeutung hat.

Selbstverständlich kann jede deklarierte Funktion nur einmal definiert werden, d. h. es kann nur einmal ein Anweisungsteil für jede Funktion geschreiben werden. Wenn eine Funktion deklariert, aber nicht definiert wird, nimmt der Compiler automatisch an, daß sie extern eingebunden werden soll. Deshalb meldet dann ggf. der Linker, nicht aber der Compiler einen Fehler.

Deklaration und Definition von Funktionen zu trennen, ist unumgänglich, wenn wie oben geschehen zwei Funktionen sich gegenseitig aufrufen. Aber auch sonst ist das oft sinnvoll: Viele Programmierer setzen an den Anfang eines Programms Deklarationen für sämtliche benutzte Funktionen. Dann darf man anschließend die Funktionedefinitionen in beliebiger Reihenfolge anordnen, muß also nicht zuerst die simplen Hilfsfunktionen und dann die gehaltvolleren Funktionen, die dann erstere benutzen, schreiben.

Funktionsdeklarationen ohne Definition benutzt man auch gerne in sog. „Header-Files" bei Projekten, die aus mehreren Modulen bestehen.

1.6.5 Überladen, Default-Parameter und andere Spezialitäten

1.6.5.1 Überladen von Funktionen

Ein recht nützliches Feature von C++ soll hier auch kurz erwähnt werden (das Thema ist in Wirklichkeit so umfangreich, daß ihm später ein eigenes Kapitel gewidnet werden soll): das „Überladen" von Funktionen.

Nehmen wir einmal an, Sie benötigen in einem Programm oft die Beträge von Zahlen, und zwar von Zahlen unterschiedlichster Datentypen. In gewöhnlichem C würden Sie dann folgendes schreiben:

```
int abs_i(int n)
{ if (n >= 0) return n;
  else return -n;
}
```

```
short abs_s(short n)
{ if (n >= 0) return n;
  else return -n;
}

float abs_f(float x)
{ if (x >= 0) return x;
  else return -x;
}

double abs_d(double x)
{ if (x >= 0) return x;
  else return -x;
}
```

Im nachfolgenden Programm müssen Sie dann immer solche Namen wie „abs_f" benutzen. Dabei wäre es doch viel schöner, wenn man immer nur „abs" sagen und der Compiler dann die richtige Funktion selbst heraussuchen würde. Und genau das ist in C++ möglich: es ist erlaubt, eine Funktion mit mehreren unterschiedlichen Parameterlisten zu deklarieren. Das könnte dann so aussehen:

```
int abs(int n)
{ if (n >= 0) return n;
  else return -n;
}

short abs(short n)
{ if (n >= 0) return n;
  else return -n;
}

// usw.
```

Wir haben es jetzt also mit mehreren Funktionen zu tun, die alle „abs" heißen. Beim Aufruf der Funktion „abs" betrachtet der Compiler dann den Typ des Arguments und ermittelt die „abs"-Funktion, zu der der Argumenttyp am besten paßt. Natürlich existieren präzise standardisierte Regeln, die festlegen, was in diesem Sinne „am besten passen" bedeutet. Diese Regeln möchte ich Ihnen an dieser Stelle aber noch nicht zumuten, zumal wir bis jetzt noch bei weitem nicht alle Datentypen kennen. Vorerst sollten Sie beim Aufruf einer überladenen Funktion darauf achten, daß die Typen der Argumente genau mit den Parametertypen der gewünschen Funktion übereinstimmen.

Ein Beispiel:

```
void f(int, double);

void f(double, int);

main()
{ f(1, 2);      // Fehler
  f(1.0, 2.0);  // Fehler
  f(1, 2.0);    // OK
}
```

Bei den beiden ersten Funktionsaufrufen wird der Compiler einen Fehler melden, denn er weiß nicht, welche Funktion hier jeweils gemeint ist. Der dritte Funktionsaufruf hingegen ist erlaubt, denn hier hat der erste Parameter den Typ „int" und der zweite den Typ „double", so daß klar ist, welche der beiden Funktionen gemeint ist.

Es ist nicht möglich, den Ergebnistyp einer Funktion zu überladen, d. h. haben zwei Funktionsdeklarationen gleichen Namen UND gleiche Parameter, so müssen auch die Ergebnistypen gleich sein. Folgendes wäre also NICHT erlaubt:

```
int f();

double f();  // Hier meldet der Compiler einen Fehler

main()
{ int i = f();
  double d = f();
}
```

Es ist zwar intuitiv klar, daß beim ersten Funktionsaufruf die „int"-Variante und beim zweiten die „double"-Version der Funktion „f" gemeint sind. C++ ermittelt die gewünschte Funktionsinstanz aber ausschließlich anhand der Argumente, ohne den „gewünschten" Ergebnistyp zu beachten. Der Grund ist einfach der, daß die Regeln für den Aufruf überladener Funktionen auch so schon kompliziert genug sind und sie niemand mehr so recht verstehen würde, wenn auch noch der Ergebnistyp in die Regel einginge.

Wie versprochen, verschone ich Sie vorerst noch mit den genauen Regeln für das Überladen von Funktionen. Wenn Sie beim Aufruf immer darauf achten, daß die Argumenttypen exakt mit den Parametertypen übereinstimmen, können Sie so viel gar nicht falsch machen. Zu guter Letzt sei noch darauf hingewiesen, daß man den Compiler in früheren C++-Versionen (1.0-Standard) eigens darauf hinweisen mußte, wenn man beabsichtigte, eine Funktion zu überladen. Das geschah dann mittels des Schlüsselworts „overload", z. B. mußte man

```
overload Funktionsname;
```

deklarieren, bevor man die Funktion dieses Namens überladen durfte. Inzwischen ist C++ nicht mehr so kleinlich, und diese expliziten „overload"-Deklarationen sind auch gar nicht mehr Bestandteil des Sprachstandards. MaxonC++ akzeptiert solche Deklarationen immer noch, damit man ältere Quelltexte problemlos übersetzen kann, es ist aber nicht empfehlenswert, derartige Deklarationen heute noch zu verwenden.

1.6.5.2 Abkürzen durch Default-Argumente

Am Beispiel des Überladens von Funktionen haben Sie sicher schon erkannt, daß C++ eine wesentlich mächtigere Programmiersprache als C ist und daß sich der Umstieg auf jeden Fall lohnt. In diesem Abschnitt soll von noch einem praktischen Feature von C++ die Rede sein, nämlich Default-Parametern.

Manchmal hat eine Funktion so etwas wie einen „kanonischen" Parameter, dem fast immer dasselbe Argument übergeben wird. Dann nervt es ganz einfach, diesen Parameter bei jedem Aufruf anzugeben, nur weil an ein paar anderen Stellen ein anderer Wert dafür benötigt wird. Dann hat der C++-Programmierer gut Lachen und benutzt einfach einen Default-Parameter.

Man deklariert einen Default-Wert für ein Argument einer Funktion, indem man in der Funktionsdeklaration hinter den Parameter ein „=" und den gewünschten Ausdruck (egal, ob nur eine Konstante oder eine komplexe Berechnung) setzt. Ein Beispiel:

```
void f(int i1, int i2 = 0);
```

Die so deklarierte Funktion kann dann wahlweise mit einem oder zwei Argumenten aufgerufen werden. Wenn nur ein Argument angegeben wird, z. B.

```
f(42);
```

setzt der Compiler für den zweiten Parameter automatisch „0" ein.

Es dürfen stets nur für den letzten bzw. die letzten Parameter einer Funktion Default-Argumente deklariert werden. Ungültig wäre deshalb

```
int f2(int a, short b=17, char c);  // Error
```

Aus technischen Gründen darf jedes Default-Argument auch nur einmal deklariert werden, auch wenn die Funktion mehrfach deklariert wird. Also wäre folgender Programmcode fehlerhaft:

```
void myfun(double = 1.0);// OK

void myfun(double = 1.0)  // Error: Redefinition of default
argument
{ // mach' irgendwas;
}
```

Korrekt wäre hingegen überraschenderweise, bei der zweiten Deklaration das Default-Argument einfach wegzulassen. Der Compiler merkt sich dann das Argument der ersten Deklaration.

Welchen Sinn hat diese Regelung? Wäre es nicht übersichtlicher und leichter verständlich, Default-Argumente bei jeder Deklaration neu anzugeben?

Das wäre tatsächlich schöner, und einige frühe C++-Implementierungen erlauben dies auch (MaxonC++ aber nicht). Das Problem ist aber, daß man jedesmal dasselbe Argument angeben müßte, und es ist eine verblüffende Tatsache, daß es nicht so einfach ist, zu definieren, wann zwei Ausdrücke „gleich" sind. Nehmen wir einmal an, im obigen Beispiel hätten wir als Default Argument zunächst „1.0/3.0" gewählt. Bei der zweiten Deklaration geben wir nun „0.333333" an. Sind diese beiden Ausdrücke gleich? Auf einem Rechner mit bis zu 6-stelliger Rechengenauigkeit sind sie es, bei mehr als 6 Stellen Genauigkeit aber nicht. Noch komplizierter wird das Ganze dadurch, daß man als Default Parameter nicht nur Konstanten, sondern nahezu beliebige Ausdrücke deklarieren kann. Sind z. B. die Ausdrücke „a+(b+1)" und „(a+b)+1" identisch? Jedenfalls würde eine verbindliche Definition, wann zwei Ausdrücke identisch sind, ziemlich schwierig. Also hat man es dabei belassen, jedes Default Argument nur einmal angeben zu können.

Dafür darf man aber auch nachträglich zusätzliche Default-Argumente deklarieren. Ein Beispiel:

```
void seltsam(int, char, double);

void seltsam(int, char, double = 1.0);

void seltsam(int, char = '?', double);

void seltsam(int = 42, char, double);
```

Nach der ersten Deklaration benötigt „seltsam" noch drei Argumente. Nun wird nachträglich ein Default für das letzte Argument deklariert, so daß wir die Funktion nun auch mit nur zwei Argumenten aufrufen können. Bei der dritten Deklaration bekommt auch noch der „char"-Parameter einen Defaultwert. Man beachte, daß hier scheinbar ein Default-Argument für einen Parameter, der nicht am Ende steht, deklariert wird, und das ist bekanntlich verboten. In Wirklichkeit weiß der Compiler natürlich, daß der letzte Parameter schon ein Default-Argument besitzt, und so hat alles seine Richtigkeit.

Zu guter Letzt wird auch noch ein Default-Argument für den ersten Parameter hinzugefügt. Jetzt haben wir die freie Auswahl, ob wir „seltsam" nun kein, ein, zwei oder drei Argumente übergeben.

Eine kleine Einschränkung sei hier noch erwähnt: Der Ausdruck, den man als Default-Argument setzt, darf nicht Werte der anderen Argumente benutzen, und zwar (unter anderem) deshalb, weil nicht definiert ist, in welcher Reihenfolge Argumente ausgewertet werden. Wenn nun ein Default-Argument von einem anderen Argument abhängt, das noch nicht berechnet wurde, wäre das Ergebnis ja undefiniert.

Also klopft Ihnen der Compiler auf die Finger, wenn Sie so etwas versuchen:

```
int multi(int a, int b=a) // Falsch!
{ return a*b;
}
```

Die Intuition dieser Funktion ist einsichtig: Wenn sie zwei Argumente hat, berechnet sie deren Produkt, bei nur einem Argument dessen Quadrat. Falls man auf derartige „Default-Argumente" absolut nicht verzichten will, kann man sie durch Überladen der Funktion „simulieren":

```
int multi(int a, int b)
{ return a*b;
}

int multi(int a)
{ return multi(a,a);
}
```

Das kostet zwar etwas mehr Laufzeit (bei nur einem Argument wird das Ergebnis nicht direkt berechnet, sondern noch eine zweite Funktion aufgerufen), aber es funktioniert.

1.6.5.3 Variable Parameterlisten

C++ hat von C auch durchaus ein paar nützliche Erbschaften übernommen, die zwar nicht schön, aber trotzdem hin und wieder praktisch sind. Dazu gehören die variablen Parameterlisten, mit denen man einer Funktion beliebig viele Parameter beliebigen Typs übergeben kann. Dazu deklariert man als letzten Parameter einer Funktion die Ellipse „...":

```
void int_und_dann_egal(int i, ...);
```

Diese Funktion erwartet als erstes Argument ein „int", und danach kann man beim Aufruf Beliebiges setzen, z. B.

```
int_und_dann_egal(17, 4.0, 'x', 93L, 0x686b);
```

oder auch ganz kurz:

```
int_und_dann_egal(2);
```

Natürlich haben die zusätzlichen Parameter dann für die Funktion keinen Namen, so daß die Funktion darauf nicht direkt zugreifen kann, sondern nur mit Hilfe der Funktionen, die in „<stdarg.h>" deklariert werden. Näheres über dieses (und alle anderen) Includefile erfahren Sie im Referenzteil dieses Handbuchs.

1.6.6 Anachronismen aus der Mottenkiste

1.6.6.1 Umschalten in den C-Modus

Im uralten „Kernigham & Richie"-Standard für die Programmiersprache C waren Parameterlisten von Funktionen ausgesprochen seltsam definiert, und leider wirkt sich das noch bis in ANSI C fort. Bei der Einführung von C++ hat man diese alten Zöpfe lobenswerterweise abgeschnitten, aber das ist dann auch die Hauptursache für Inkompatiblitäten zwischen ANSI C und C++.

Diese „Erbschaften" von Uralt-C sind so verstaubt und sinnlos, daß sie von MaxonC++ (in Übereinstimmung mit dem C++-Standard) nicht mehr unterstützt werden, jedenfalls nicht im C++-Modus. Allerdings sind sie immer noch Bestandteil des ANSI-C-Standards, übrigens mit gutem Grund, denn es haut mich immer wieder um, wenn ich sehe, wie viele C-Programmierer diese antiquierten Features noch benutzen.

Jedenfalls möchte ich Sie noch einmal darauf hinweisen, daß es wirklich keinen vernünftigen Grund gibt, das im folgenden Beschriebene zu benutzen. Ich schreibe dieses Kapitel überhaupt aus nur drei Gründen:

1. MaxonC++ kann es, also soll es auch dokumentiert werden.
2. Man braucht diese alten Features, wenn man einen älteren Quelltext mit MaxonC++ übersetzen will.
3. Vielleicht versuchen Sie einmal, ein fremdes Listing zu verstehen, und dann sollen Sie sich nicht über eine gewisse seltsame Syntax wundern.

Bevor ich Ihnen verrate, worum es eigentlich geht, erzähle ich Ihnen noch kurz, wie Sie unter MaxonC++ in den ANSI-Modus gelangen. Bestimmt haben Sie schon das Pull-Down-Menü „Optionen/Compiler/C++" entdeckt. Wenn man diesen Menüpunkt aus-

schaltet, wird das Programm nach den ANSI-C-Sprachregeln übersetzt. Aber man kann auch im Programmtext den Sprachstandard wählen und sogar an beliebiger Stelle zwischen C und C++ umschalten, und zwar mit einem „pragma".

„#pragma"-Zeilen werden, genau wie „#include", vom Preprozessor interpretiert und ausgeführt. Mit einem „#pragma" realisiert man dabei compiler-spezifische Features, die ausdrücklich nicht standardisiert sind. In MaxonC++ schaltet die Zeile

```
#pragma -
```

den Compiler auf ANSI-C-Emulation um, und

```
#pragma +
```

ermöglicht danach wieder die Programmierung in C++.

Jeder C-Compiler, der zu ANSI C kompatibel ist, kennt übrigens „pragma"-Zeilen, nur das, was dahinter steht und was es bedeuten soll, ist nicht im C-Standard festgelegt, denn damit soll man ja eben implementationsabhängige Funktionen ansprechen. Deshalb ist festgelegt, daß der Compiler keinen Fehler melden darf, wenn er auf ein unbekanntes „pragma" trifft. Also sollte jeder andere ANSI-C-Compiler ein „#pragma -" aus einem unter MaxonC++ entwickelten Programm ignorieren, während MaxonC++ Dinge wie „#pragma Hallo!" ganz einfach nicht beachtet.

1.6.6.2 Parameterlisten im alten Stil

Und nun lasse ich die Katze aus dem Sack: Bei den oben erwähnten Uralt-Features handelt es sich um die alte Syntax für Parameterlisten und die Konsequenzen, die das für ANSI C bis heute hat.

Ursprünglich wurden die Datentypen von Argumenten überhaupt nicht beachtet. Rief man eine Funktion auf, etwa

```
f(17, 4.0);
```

so legte der Compiler die „int"-Zahl 17 und die „double"-Konstante „4.0" auf den Stack und rief dann die Funktion „f" auf, ganz egal, was für Argumente „f" in Wirklichkeit erwartet. Das können Sie heute auch noch in C++ simulieren, indem Sie „f" mit einer vollkommen variablen Argumentenliste deklarieren:

```
int f(...);
```

In jenen alten Tagen mußte man eine Funktion dann auch überhaupt nicht deklarieren, bevor man sie aufrief - die Argumenttypen waren ja sowieso egal, und wenn nichts

anderes deklariert wurde, wurde als Ergebnistyp „int“ angenommen. Wenn man eine Funktion aber trotzdem deklarierte, sah das ganz einfach so aus:

```
int f();
```

Man durfte sogar den Ergebnistyp weglassen, also

```
f();
```

deklarieren, wobei dann als Ergebnis im Zweifelsfall „int“ angenommen wurde. Eine solche Deklaration hat natürlich niemand geschrieben, denn da wird ja nichts anderes gesagt, als der Compiler ohnehin schon vermutet, nämlich daß die Funktion „f“ existiert, irgendwelche nicht näher bekannten Argumente hat und entweder „int“ oder gar nichts zurückgibt.

Bei der Definition der Funktion war es dann aber nötig, den Parametern Namen zu geben. Das sah dann so aus, daß nur die Namen in die Parameterliste gesetzt wurden. Zwischen der Funktionsdeklaration und dem Funktionsrumpf konnte man dann die Datentypen der Parameter deklarieren - wenn für einen kein Typ angegeben wurde, so wurde er als „int“ betrachtet.

Ein Beispiel:

```
#pragma -   /* Das folgende akzeptiert MaxonC++ nur im ANSI-Modus */

int uralt(i,j,k,d,c)
  int i, j;
  double d;
  char c;
{
  d = i+j+k;
  return (int)d + c;
}
```

Aber glauben Sie nicht, der Compiler hätte wenigstens nach dieser Funktionsdefinition die Argumenttypen beachtet, oh nein! Auch danach blieben die Typen der Parameter Privatangelegenheit der Funktion, und beim Funktionsaufruf konnte der Programmierer nur beten, daß er immer die richtigen Datentypen erwischte, sonst bekam die Funktion unsinnige Parameter. Ach ja, das waren noch Zeiten, als Programmierer noch echte Männer waren und souverän die Fallgruben, die C ihnen grub, umgingen...

Übrigens kamen in letzter Zeit Gerüchte auf, daß das alte C eigentlich nur ein Scherz sein sollte. Demnach wollten Kernighan und Richie eine möglichst konfuse Parodie auf Pascal schreiben und das Ganze dann den Russen unterjubeln, um deren technologischen Rückstand weiter zu vergrößern. Ebenso soll UNIX ursprünglich nur eine Parodie auf

MULTICS gewesen sein, aber dann kamen einige Verrückte und begannen, unter UNIX in C tatsächlich zu programmieren... Aber lassen wir das.

1.6.6.3 Von der Steinzeit in die ANSI-Ära

Mit der Einführung von ANSI C wurde dann einiges besser. Endlich wurden die Parameterlisten, die man schon von C++ kannte, in C aufgenommen. Sie haben durchaus richtig gelesen, denn C++ gab es schon vor ANSI C, wenn auch in einer etwas abgespeckten Version. Dabei gab es dann aber ein kleines Problemchen: das Ganze sollte auf jeden Fall kompatibel bleiben. Also blieben die alten Parameterlisten 'drin, und der Programmierer hat in ANSI C bis heute die Wahl, ob er nun im neuen oder alten Stil schreiben will. Es wird zwar allgemein empfohlen, die neue Form zu benutzen (was ja auch das einzig Sinnvolle ist), aber das interessiert erfahrungsgemäß den echten Hard-Core-C-Coder nicht.

Um die Kompatiblität zum alten C-Standard zu erhalten, mußte man definieren, daß die scheinbar leere Parameterliste „()" weiterhin für „beliebige Parameter ohne Typprüfung" stehen sollte - also ein deutlicher Unterschied zu C++. Eine Funktion, die wirklich keine Parameter haben soll, deklariert man in ANSI C, indem man in die Parameterliste „void" schreibt, z. B.

```
int fun(void);
```

Das sieht komisch aus und nervt, deshalb ist es allgemein verbreitet, für parameterlose Funktionen einfach „()" zu schreiben und dann selbst darauf zu achten, daß man keine überflüssigen Argumente übergibt.

MaxonC++ akzeptiert die „(void)"-Schreibweise auch im C++-Modus, damit man leichter gemeinsame Header-Files für C und C++ schreiben kann. Dem C++-Standard entspricht dies genaugenommen nicht, aber es ist dem Implementator freigestellt, derartige Anachronismen einzubauen. Wenn man unter C++ Parameterlisten im alten Stil benutzt, meckert MaxonC++ und bittet darum, in den C-Modus umzuschalten.

Zusammenfassend möchte ich zusammenfassenderweise zusammenfassen, daß die leere Parameterliste „()" je nach Modus entweder „(...)" oder „(void)" entspricht.

1.6.7 Einige Standardfunktionen

1.6.7.1 Allgemeines

Der C-Programmierer soll das Rad nicht jedesmal neu erfinden. Deshalb gehört zu jedem C-Entwicklungssystem eine umfangreiche Sammlung von Bibliotheksfunktionen für

alle Lebenslagen. Diese werden im Referenzteil dieses Handbuchs ausführlich und systematisch beschrieben. Deshalb soll dieser Abschnitt Ihnen nur einen kleinen Überblick geben, damit Sie bei Bedarf besser Bescheid wissen, wo Sie zu suchen haben.

C hat im Gegensatz zu Pascal, Basic und anderen Programmiersprachen keine „eingebauten" Funktionen (in Pascal sind Funktionen wie „writeln" oder „abs" von Anfang an vorgegeben). In C ist das so konzipiert, daß man immer Includefiles (so wie „stream.h"), in denen dann die Funktionsdeklarationen stehen, benutzen muß. Der Linker bindet dann die benutzten Funktionen ein.

Erfreulicherweise gehören zur Sprachdefinition von ANSI C auch genormte Bibliotheken, so daß C-Programme theoretisch problemlos auf andere Rechner portiert werden können. Bei C++ ist man leider noch nicht so weit: hier gibt es nur als Konvention die Datei „stream.h", deren Inhalt aber so genau gar nicht festgelegt ist.

Außerdem gibt es noch ein paar Spezialfunktionen von MaxonC++ und last not least die Amiga-Betriebssystemfunktionen. In diesem Abschnitt soll aber vor allem von den Standardfunktionen die Rede sein.

1.6.7.2 Ein- und Ausgabe: <stdio.h>

Die Includedatei „stdio.h" deklariert die Funktionen, die unter C zur Ein- und Ausgabe von Daten dienen (da man in C keine Operatoren überladen kann, fallen „cout" und „cin" hier flach).

Die wohl bekannteste C-Funktion ist „printf", wobei Sie das „print" wahrscheinlich schon aus BASIC-Tagen kennen und das „f" für „Format" steht. Also dient „printf" zur universellen formatierten Ausgabe von Daten.

Als ersten Parameter erwartet „printf" einen sog. Formatstring. Das ist eine beliebige Zeichenkette, die dann im Prinzip auch ausgegeben wird, nur daß das Zeichen '%' dabei eine besondere Bedeutung hat: '%', gefolgt von Formatangaben und einem Buchstaben, fügt an dieser Stelle des Strings Daten ein, die man hinter dem Formatstring angibt.

Ein Beispiel:

```
#include <stdio.h>

void main()
{ int a = 17, b = 4;
  printf("Die Summe von %d und %d ist: %d\n", a, b, a+b);
}
```

Wo immer „%d" in der Zeichenkette auftritt, nimmt „printf" das nächste Element aus der Parameterliste und gibt es an dieser Stelle als „int" aus. Dabei gibt es noch etliche Variationen: Bei „%x" erfolgt die Ausgabe in Hexadezimal, „%hd" bzw. „%hx" geben entsprechend „short"- oder „unsigned short"-Daten aus, „%e", „%f" und „%g" geben „double"-Zahlen in unterschiedlichen Formaten aus usw. Dabei programmieren Sie übrigens ohne Netz und doppelten Boden, denn die Typen der Parameter hinter dem Formatstring werden nicht geprüft. Wenn Sie also einen „short int"-Wert mit „%d" statt „%hd" ausgeben, werden Sie unsinnige Ausgaben erhalten.

Alles in allem ist „printf" eine sehr mächtige Funktion, die weitaus flexiblere Ausgaben als „cout" ermöglicht, manchmal aber auch etwas komplizierter zu handhaben ist.

Es gibt dann noch die Funktion „scanf" für formatierte Eingaben. Ihr muß man Zeiger auf die einzulesenden Variablen übergeben, aber Zeiger werden erst im nächsten Kapitel behandelt.

Vor allem enthält „stdio.h" aber die gesamte Dateibehandlung. Die Funktion „fopen" öffnet eine Datei, „fclose" schließt sie, und mit „fprintf" und „fscanf" kann man formatiert in eine Datei schreiben bzw. aus ihr lesen.

1.6.7.3 Zeichen, Zeichenketten und Bytefolgen:

<ctype.h> und <string.h>

Die Datei „ctype.h" deklariert kleine Funktionen auf Zeichen. Beispielsweise liefert die Funktion „isalpha(c)" den logischen Wert „wahr", wenn ihr Argument ein Buchstabe ist. Entsprechend stellen „islower(c)" und „isupper(c)" fest, ob ihr Argument ein Klein- bzw. Großbuchstabe ist. Entsprechende Funktionen gibt es auch für den Test auf Ziffer, Steuerzeichen und anderes.

Diese Funktionen sind vor allem deshalb wichtig, weil auch heute nicht jede (aber fast jede) Maschine den ASCII-Code benutzt. In ASCII könnte man

```
if (isupper(c))
```

ersetzen durch

```
if (c >= 'A' && c <= 'Z')
```

Das muß aber auf Nicht-ASCII-Systemen keineswegs so sein. Auf diese Weise verbessern die <ctype.h>-Funktionen die Portabilität von Programmen.

Nützlich sind auch „tolower(c)“ und „toupper(c)“: Erstere wandelt ihr Argument, wenn es ein Großbuchgstabe ist, in einen Kleinbuchstaben und gibt es andernfalls unverändert zurück, letztere wandelt entsprechend Klein- in Großbuchstaben.

In <string.h> stehen die elementaren Funktionen für die Stringbehandlung von ANSI C, z. B. Kopieren, Aneinanderhängen oder Vergleichen von Zeichenketten. Zur näheren Erläuterung dieser Funktionen müßten Sie sich aber mit Zeigern und Vektoren auskennen, aber das steht erst im nächsten Kapitel.

Aus selbigem Grund sollen die „mem“-Funktionen, die ebenfalls in „string.h“ deklariert werden, hier nur kurz erwähnt werden. Sie stellen Operationen auf Speicherblöcken dar.

1.6.7.4 Fließkommafunktionen: <math.h>

Wenn Sie die Includedatei „<math.h>“ in Ihr Programm aufnehmen, stehen Ihnen die wichtigsten mathematischen Funktionen zur Verfügung. Vor allem werden dort „double“-Funktionen wie

```
double sin (double)
```

deklariert, die - Sie werden es erraten haben - den Sinus ihres Arguments berechnet. Entsprechend berechnen die Funktionen „cos“ den Cosinus und „tan“ den Tangens. Bei diesen trigonometrischen Funktionen sind Winkel stets im Bogenmaß (Radian) anzugeben, d. h. 360 Winkelgrad entsprechen 2*Pi.

Natürlich gibt es hier neben vielen anderen auch die Funktionen „exp“ (für die Exponentialfunktion), „log“ (natürlicher Logarithmus), „sqrt“ (Quadratwurzel) und „fabs“ (Betrag). „floor“ rundet ab (liefert also die nächstkleinere ganze Zahl), wobei das Ergebnis aber trotzdem ein „double“-Wert ist, und „ceil“ rundet entsprechend auf.

MaxonC++ bietet in „<math.h>“ außerdem noch einige Funktionen, die Zahlen in ihre Zeichendarstellung verwandeln.

1.6.7.5 Die C-Fundgrube: <stdlib.h>

Alle wichtigen Funktionen, die sich unter keine der bisherigen Rubriken einordnen ließen, wurden in <stdlib.h> untergebracht. Besonders wichtig ist hier wohl die Funktion

```
void exit(int)
```

Sie bricht die Programmausführung ab, als wäre das Ende der Funktion „main“ erreicht worden, und gibt eine Zahl (z. B. einen Fehlercode) an das aufrufende System zurück.

Häufig braucht man auch die Funktionen „abs“ und „labs“, die den Betrag eines „int“- bzw. „long int“-Parameters berechnen. Wie Sie sehen, beherrscht ANSI C kein Überladen von Funktionen...

Praktisch ist auch die Funktion „rand()“, die eine ganzzahlige Zufallszahl im Bereich von 0 bis 32767 berechnet. In Wirklichkeit handelt es sich dabei nicht um Zufallszahlen, sondern um eine Zahlenfolge, die nur so aussieht, als wäre sie zufällig. Deshalb ist es auch nötig, den Startwert dieser Folge vor der ersten Benutzung von „rand“ mit der Funktion „srand(unsigned int)“ zu setzen, meist benutzt man hier die Systemzeit oder etwas Ähnliches.

MaxonC++ bietet Ihnen hier noch zwei andere Funktionen, oder sagen wir besser: eine überladene Funktion:

```
double Random()
```

liefert eine Fließkomma-Zufallszahl im Bereich von 0 bis 1, und

```
unsigned int Random(unsigned int n)
```

berechnet eine Zufallszahl im Bereich von 0 bis n-1. So kann man z. B. mit „Random(6)+1“ einen Würfel simulieren. Die beiden „Random“-Funktionen benutzen nicht nur ein Standardverfahren, sondern als zusätzliches Zufallselement die aktuelle Rasterposition des Elektronenstrahls. Deshalb müssen sie auch nicht mit „srand“ initialisiert werden.

Außerdem enthält „stdlib.h“ Funktionen, die Zeichenketten in Zahlen verwandelt, und die ANSI-C-Funktionen zur Speicherverwaltung (z. B. „malloc“). Letztere dürfen Sie aber getrost vergessen, denn in C++ geht das viel eleganter. Nützlich ist auch die Funktion „qsort“, die ein Datenfeld nach dem Quicksort-Verfahren sortiert.

Dieser Abschnitt hat Ihnen hoffentlich einen kleinen Überblick über die Funktionen von ANSI C gegeben. Alles weitere finden Sie im Referenzteil dieses Handbuchs.

2. Datentypen und Deklarationen

2.1 Vektoren, Zeiger und Referenzen

2.1.1 Pointer

2.1.1.1 Von Objekten und Adressen

Ein zentrales Konzept der Programmierung in C (und Quelle ewiger Verzweiflung für Anfänger) sind die Zeiger, auf neudeutsch auch „Pointer" genannt. Um in diesem Zusammenhang Konfusion von Anfang an zu vermeiden, sollten wir uns erst einmal genauer ansehen, was im Speicher so los ist.

Eine Variable in C ist ein Datenobjekt, das irgendwo im Speicher liegt. Wahrscheinlich (hoffentlich) wissen Sie, wieviel Bytes RAM Ihr Amiga hat. Damit der Prozessor damit auch etwas anfangen kann, sind alle diese Bytes durchnumeriert - das nennt man die „Adresse" eines Speicherbytes. Beim 68000-er Prozessor, der (eventuell in verbesserter Ausführung) in jedem Amiga steckt, ist eine Adresse 32 Bit breit, genau wie z. B. ein „int" in MaxonC++. Theoretisch kann man damit also gediegene 4 Gigabyte Speicher adressieren (Adressen betrachtet man übrigens normalerweise als vorzeichenlos). Jeder real existierende Amiga hat natürlich weniger RAM, so daß viele Adressen nicht benutzt werden. Trotzdem ist eine Adresse stets 32 Bit groß.

Nun ist ein Byte nicht gerade viel, nämlich gerade genug, um darin ein Zeichen (z. B. eine „char"-Variable) abzulegen. „Breitere" Datentypen nehmen dann entsprechend mehrere Bytes ein, die stets an aufeinanderfolgenden Adressen liegen. Als Adresse dieses größeren Datengebildes gilt dabei stets die Adresse des ersten Bytes. Wenn also ein „int"-Objekt die vier Speicherbytes von 0x00053ac4 bis einschließlich 0x00053ac7 belegt (Adressen gibt man gerne als hexadezimale Zahlen an), sagt man einfach, es liegt an Adresse 0x00053ac4.

Nun könnte man ganz einfach eine Speicheradresse eines Objekts in einer Variablen des Typs „unsigned int" abspeichern und dies dann als Verweis oder Zeiger auf die Speicherstelle benutzen. Weil das Zeigerkonzept aber so enorm wichtig und nützlich ist, wird es von C noch viel weitreichender unterstützt: Man kann sich zu jedem Datentypen „X" einen Pointertypen „Zeiger auf X" deklarieren. Der unäre Operator „&", der einem Objekt vorangestellt wird, liefert die Adresse dieses Objekts, die dann automatisch vom entsprechenden Pointertyp ist. Mit dem unären „*"-Operator erhält man dann umgekehrt wieder aus einem Zeigerausdruck das dadurch „referierte" Datenobjekt.

Klingt das kompliziert? Nun, dann wollen wir gleich zur Praxis übergehen.

2.1.1.2 Zeiger: Deklaration und elementare Operationen

Wie oben bereits angedeutet, gibt es nicht nur einen einzigen Datentyp „Zeiger", sondern zu jedem einzelnen Datentyp einen entsprechenden Pointertyp. Folglich gibt es auch „Zeiger auf Zeiger" und „Zeiger auf Zeiger auf Zeiger auf...", also unendlich viele Datentypen. Natürlich kennt der Compiler nicht unendlich viele vordefinierte Namen für alle diese Typen. Deshalb müssen wir hier erstmals einen Datentypen selbst definieren, indem wir ihn beschreiben.

Ein (übrigens nicht sehr schlaues) Prinzip der C-Syntax ist, daß die Deklaration einer Variablen möglichst genau wie ihre Benutzung aussehen soll. Nehmen wir an, „p1" soll ein Zeiger auf einen „double"-Wert sein. Dann ist, wie oben bereits angedeutet, „*p1" eben das „double"-Wort, auf das „p1" gerade zeigt. Deshalb deklarieren wir die Zeigervariable „p1" folgendermaßen:

```
double *p1;
```

Das ist relativ einleuchtend: wir deklarieren nämlich, daß „*p1" vom Typ „double" sein soll, folglich ist „p1" ein Zeiger auf „double". Man sagt auch: p1 ist vom Datentyp „double*". Deshalb sollten Sie aber nicht in Versuchung geraten, mehrere Pointervariablen so zu deklarieren:

```
int* i1, i2;  // Vorsicht!
```

Hier gehört das „*" ausschließlich zu „i1". Demnach deklariert diese Zeile „i1" als Zeiger auf „int" und „i2" als „int". Mehrere Zeiger muß man so deklarieren:

```
int *i1, *i2, *i3;
```

Unmittelbar nach der Deklaration ist ein Zeiger, genau wie alle anderen Variablen auch, undefiniert und zeigt „irgendwohin". Bevor man ihn benutzt, sollte man ihm also unbedingt einen sinnvollen Wert geben, z. B. auf eine andere Variable zeigen lassen. Das könnte so aussehen:

```
#include <stream.h>

void main()
{ int i1 = 41, i2 = 98; // Zwei ganz normale int-Variablen

  int *ip = &i1;        // Jetzt zeigt ip auf i1

  cout « *ip;           // Gibt den Wert von i1 aus

  i1++;                 // i1 wird verändert
```

```
    cout « *ip;              // Jetzt wird der NEUE Wert ausgegeben

    ip = &i2;                // Nun zeigt ip auf i2

    *ip = 26731;             // Wert von i2 wird über ip geändert

    cout « i2;               // Also Ausgabe "26731"
}
```

Der Operator „&" liefert die Adresse eines Objekts, wobei das Ergebnis den jeweiligen Pointertyp hat. Also ist im obigen Beispiel „&i1" ein Ausdruck des Typs „int*" und somit eine passende Initialisierung für die Variable „ip". Nun zeigt „ip" also bis auf weiteres auf „i1", und wir können „*ip" synonym zu „i1" verwenden. Wenn wir also „*ip" mit „cout" ausgeben, wird immer der Wert ausgegeben, denn „i1" gerade hat, nicht etwa der, den „i1" im Zeitpunkt der Zuweisung hatte.

Nicht nur der Wert des Objekts, auf das ein Pointer zeigt, sondern auch der Wert eines Pointers selbst kann verändert werden. Um das zu demonstrieren, wird im Beispielprogramm „ip" anschließend die Adresse von „i2" zugewiesen. Und zu allem Überfluß wird danach auch noch der Wert von „*ip" verändert! Da „ip" jetzt aber auf „i2" verweist, ist klar, was das bedeutet: nun ändert sich der Wert von „i2", ohne daß wir diesen Variablennamen auch nur erwähnen!

Daß Variablen auf diese Weise unter einem anderen Namen benutzt werden können und man außerdem noch zwischen einer Zeigervariablen und dem von ihr referierten Objekt unterscheidet, ist für Anfänger oft so verwirrend, daß sie beschließen, auf Pointer ganz zu verzichten und so viele Dinge niemals realisieren können. Ich empfehle Ihnen dringend, sich nicht abschrecken zu lassen, wenn man sich einmal etwas daran gewöhnt hat, ist das alles absolut trivial.

2.1.1.3 Zeiger als Funktionsparameter

In C benutzt man Zeiger gern, um Variablenwerte von Funktionen verändern zu lassen, denn C kennt (im Gegensatz zu C++) keine „VAR-Parameter" wie Pascal. Ein kleines Beispiel:

```
#include <stream.h>

void Quadriere (int *par)
{
  *par = *par**par;  // Schlechter Stil, übersichtlicher wäre:
                     // *par = (*par) * (*par)
}
```

```
main()
{ int i = 17;

  Quadriere (&i);     // Wir übergeben nicht "i", sondern einen
                      // Zeiger darauf!
  cout « "17^2 ist: " « i;
}
```

Die Funktion „Quadriere" ist als „void" deklariert, liefert also keinen Wert direkt zurück. Vielmehr erwartet sie als Argument einen Zeiger auf ein „int"-Objekt, dessen Wert sie quadriert und dann in eben jenem Objekt ablegt. Die chaotische Ansammlung von „*"-chen (teils ist der unäre Inhaltsoperator, teils der binäre Multiplikationsoperator gemeint) soll hier nur illustrieren, daß man ein Programm mit Pointeroperationen leicht unübersichtlich gestalten kann. In der Praxis sollte man wohl besser Klammern zur Gliederung verwenden.

Beim Aufruf der Funktion muß man dann natürlich nicht die Variable selbst, sondern einen Zeiger darauf als Argument übergeben. Diese Art der Parameterübergabe ist also erheblich umständlicher als ein VAR-Parameter in Pascal. Aber keine Panik: C++ bietet, als Erweiterung gegenüber C, Referenztypen, mit denen man genau das viel schöner realisieren kann.

Übrigens kann eine Funktion auch einen Zeiger zurückgeben:

```
int *pfun()
{ int j = 99;
  return &j;       // Vorsicht!!!
}

void main()
{ cout « *pfun(); // Ergebnis undefiniert!
}
```

Die Funktion „pfun" liefert hier als Ergebnis einen Zeiger auf eine lokale Variable, aber nach Beendigung der Funktion existiert diese Variable nicht mehr, so daß das Funktionsergebnis hier lediglich dahin zeigt, wo die Variable „j" einmal gewesen ist. Es gibt keine Garantie, daß dieser Speicherplatz nach dem Funktionsaufruf nicht sofort wieder überschrieben wird. Also sollte man sich immer gründlich überlegen, welchen Zeiger man aus Funktionen gefahrlos hinausreichen kann.

2.1.1.4 Zeigertypen

Bisher habe ich Ihnen diskret verschwiegen, daß ein „int*" etwas ganz anderes als z. B. ein „char*" ist. Zwar werden beide intern als 32 Bit breite Adresse dargestellt, aber trotzdem ist es so ohne weiteres nicht erlaubt, unterschiedliche Zeiger einander zuzuweisen:

```
int *i, *ii;
char *c;

ii = i;          // Erlaubt, denn gleiche Typen
c = i;           // Verboten!
c = (char*) i;   // So geht's.
```

Wenn man einen Zeiger in einen anderen Zeigertyp umwandeln will, muß man das also (z. B. mit einem Cast) deutlich sagen, denn auf diese Weise kann man erheblichen Schaden anrichten - den Rechner zum Absturz zu bringen, ist nicht weiter schwierig! Immerhin ist ein „int*" ein Zeiger auf das erste von vier Bytes, während ein „char*" auf ein einzelnes Byte zeigt. Würde man im letzten Beispiel eine Zuweisung an „*c" vornehmen, so würden also das erste Byte von „*i" verändert und die drei anderen unverändert bleiben - also wäre der Wert von „*i" irgendetwas (wahrscheinlich) Sinnloses. Noch fataler wäre folgende Zuweisungsfolge:

```
// c zeigt auf irgend so ein char
i = (int*) c;
*i = 4711;  // PANIK!
```

Die Zuweisung an „*i" verändert nicht nur den Wert von „*c", sondern auch die drei folgenden Bytes, und da wir nicht wissen, ob diese drei Bytes irgendeine besondere Bedeutung haben, ist das Ergebnis völlig unbestimmt!

Es gibt aber einen besonderen Pointertypen, der für alle anderen stehen darf, nämlich „void*". Das ist, wie der Name schon sagt, ein Zeiger auf „nichts Besonderes". Ein „void"-Pointer repräsentiert zwar eine Speicheradresse, zeigt aber auf kein wirkliches Objekt, bzw. auf ein Objekt des Typs „void".

```
void main()
{ int i;
  int *ip = &i;
  void *vp;

  vp = ip;     // Erlaubt
  ip = vp;     // In C++ falsch
  *vp = 99;    // Noch falscher
}
```

Der Zeiger „ip" darf ohne explizite Typwandlung an den Voidpointer „vp" zugewiesen werden, denn mit einem „void" kann man so gut wie nichts machen, so daß für das von „ip" referierte Datenobjekt keine Gefahr besteht. Umgekehrt wird aber bei der Zuweisung „ip = vp" ein Fehler gemeldet, denn hier wird ja wieder einmal ein „Zeiger auf irgendwas" in einen ganz bestimmten Zeigertypen umgewandelt. Falls Sie das wirklich wollen, müssen Sie hier wieder einen Cast setzen.

Die letzte Zeile zeigt dann auch, warum „vp = ip" zulässig war. Es ist nicht möglich, mit der Speicheradresse, auf die ein „void*" zeigt, etwas Gravierendes anzustellen, es sei denn, man wandelt den „void"-Pointer wieder in einen anderen Zeigertyp um, z. B. so:

```
*(int*)vp = 99;   // wandle "vp" erst in ein "int*" um
```

Nebenbei bemerkt haben wir es hier wieder mit einem Unterschied zwischen C und C++ zu tun: In C ist es nämlich erlaubt, ein „void*" ohne Casting in einen beliebigen Zeigertypen umzuwandeln. Das ist wieder einmal ein Beispiel dafür, daß man in C++ viel sauberer und sicherer programmieren kann als in C.

Man kann einen Zeiger auch, wenn es unbedingt sein muß, in eine Zahl umwandeln und wieder zurück. Ein Beispiel:

```
#include <stream.h>

void main()
{ int var;
  cout « "var liegt an Adresse: " « (unsigned int)&var;
}
```

Das obige Beispiel ist übrigens nicht portabel! Es ist keineswegs garantiert, daß ein „int" und ein Zeiger gleich breit sind (in diesem Fall: beide jeweils 32 Bit haben). Auf manchen Systemen ist ein „int" 16 Bit breit, wie „short int" unter MaxonC++, so daß im Beispiel nicht die Adresse, sondern nur deren untere 16 Bit ausgegeben würde. Wenn Sie in dieser Situation nach „long" wandeln, dürfte es auf den meisten Systemen klappen, aber noch nicht einmal das ist strenggenommen garantiert. Somit dürfte klar sein, daß Umwandlungen zwischen Zahlen und Zeigern eine etwas problematische Sache sind.

Eine Ausnahme gibt es allerdings: Die konstante Zahl „0" kann direkt in jeden Zeigertyp umgewandelt werden. Das benutzt man gern, um anzuzeigen, daß der Zeiger gerade auf „nichts" zeigt.

2.1.1.5 Was man sonst noch so mit Zeigern machen kann

Man darf Zeigerausdrücke gleichen Typs auch mit den Operatoren

```
==   !=   <   >   <=   >=
```

vergleichen. Dabei werden dann die Speicheradressen verglichen, als wären sie „unsigned"-Zahlen. So kann man mit „==" bzw. „!=" feststellen, ob zwei Pointer auf dasselbe Objekt zeigen. Die anderen Operationen sind eigentlich nur dann sinnvoll, wenn die beiden Zeigerausdrücke auf Elemente desselben Vektors zeigen (siehe Abschnitt 2.1.2).

Natürlich kann man Zeiger auch mit dem ausgezeichneten Zeigerwert „0" vergleichen:

```
double *dptr;

// ...irgendwas

if (dptr == 0)
{ // usw.
```

Es wird garantiert, daß kein Datenobjekt an Adresse 0 liegt, so daß die „if"-Abfrage hier feststellt, ob „dptr" auf „nichts" zeigt.

Ein Zeigeraudruck kann auch direkt als logische Bedingung dienen und gilt als „wahr", wenn er nicht „0" ist. Also kann man die „if"-Anweisung verkürzen zu:

```
if (dptr)
{ // usw.
```

Ebenso kann ein Zeigerausdruck direkt als Operand der logischen Operatoren „!", „&&" und „||" benutzt werden.

Leider ist es möglich, auf die von einem „Nullzeiger" referierte Adresse zuzugreifen:

```
double *dp = 0;

*dp = 17.4; // VORSICHT!
```

Dieses Programmfragment dürfte den Amiga zum Absturz bringen, denn an den Adressen 0 bis 7, die hier gnadenlos überschrieben werden, liegen durchaus wichtige Betriebssystemdaten. Solche Aktionen sind, genau wie andere Operationen auf gar nicht oder sinnlos initialisierten Zeigern, deshalb möglich,

1. weil der Compiler nicht schon beim Übersetzen feststellen kann, ob eine Zeigervariable an einer bestimmten Stelle einen sinnvollen Wert hat. Ein so simples Beispiel wie das letzte, bei dem direkt nacheinander ein Zeiger auf 0 gesetzt wird und dann das referierte Objekt manipuliert wird, könnte ein schlauer Compiler zwar abfangen. Es ist aber (und das ist mathematisch beweisbar!) möglich, jeden noch so geschickt geschriebenen Compiler hinters Licht zu führen.

2. weil es einfach zu viel Rechenzeit kosten würde, zur Laufzeit des Programms vor jeder Zeigeroperation erst festzustellen, ob der Pointer auf einen sinnvollen Wert zeigt. Auch hier gilt der Grundsatz: „Der C-Programmierer weiß, was er tut."

Pointer haben vieles mit den Vektoren, die im nächsten Kapitel besprochen werden, gemeinsam. Deshalb kann man die binären Operatoren „+" und „-" sowie den Indexoperator „[ ]" darauf anwenden. Aber das gehört dann auch schon in das folgende Kapitel:

2.1.2 Vektoren

2.1.2.1 Das Vektor-Konzept in C

Praktisch jede Programmiersprache kennt so etwas wie Felder, Arrays oder Vektoren, nur die Bezeichnungen sind unterscheidlich und werden auch beliebig durcheinander verwendet. Das Prinzip ist aber immer das gleiche: Ein Datenobjekt besteht aus einer endlichen Anzahl von Elementen, die jeweils denselben Typ besitzen und durchnummeriert sind. Sowohl der Vektor als auch jedes einzelne seiner Elemente kann man als Datenobjekt auffassen.

Sei „v" ein Vektor, der aus 10 Elementen des Typs „int" besteht. Dann heißen die einzelnen Elemente

```
v[0], v[1], v[2], ..., v[8], v[9]
```

Also greift man auf ein Element eines Vektorausdrucks zu, indem man den „Index", der ein beliebig komplizierter „int"-Ausdruck sein kann, in eckigen Klammern hinter den Vektorausdruck stellt. Man beachte, daß Indizes (bitte nicht „Indexe" sagen!) in C immer von 0 an gezählt werden. Jedes dieser Vektorelemente ist dann eine eigenständige Variable (hier: ein „int"-Objekt) und kann auch als solche in Programmen verwendet werden.

Die eckigen Klammern gelten übrigens als Operator, und zwar als Postfix-Operator, der stärker bindet als z. B. die vorangestellten unären Operatoren.

Natürlich müssen auch Vektoren deklariert werden. Wieder einmal gilt hier, daß die Deklaration wie die Benutzung aussehen soll. Also wird der Vektor „v" so deklariert:

```
int v[10];      // deklariert den Vektor v[0], v[1], ..., v[9] !!!
```

Ein kleines Beispiel für Vektoren:

```
// Ein kleines Statistik-Programm

#include <stream.h>

void main()
{ int i;
  double dat[20];

  cout « "Bitte geben Sie 20 Zahlen ein:\n";

  for (i=0; i<20; i++)
  { cout « "Zahl " « i+1 « ": ";
    // C zählt von 0 bis 19, der Benutzer wahrscheinlich lieber
    // von 1 bis 20. Deshalb wird hier „i+1" ausgegeben.
    cin » dat[i];
  }

  double sum = 0;
  for (i=0; i<20; i++)     // Wie summieren die
    sum += dat[i];         // eingegebenen Zahlen

  double mittel = sum/20;  // Mittelwert = Summe/Anzahl

  cout « "\nSumme:      " « sum;
  cout « "\nMittelwert: " « mittel;
  cout « "\n";
}
```

Leider muß die Größe eines Vektorobjekts immer eine Konstante sein. Wenn das Programm dahingehend verbessert werden soll, daß der Benutzer wählen kann, wie viele Zahlen er einzugeben gedenkt, bleibt uns (vorerst) nichts anderes übrig, als einen ziemlich großen Vektor (vielleicht „dat[1000]") einzurichten und dann nur die benötigten Elemente davon zu benutzen.

C prüft nicht, ob ein Index im „erlaubten" Bereich liegt! Hätte man oben nur „dat[10]" deklariert und dann trotzdem 20 Elemente benutzt, wäre das Ergebnis „undefiniert", was in der C-Sprachdefinition der übliche Euphemismus für „Absturz" ist. Auch hier gilt einmal wieder: „Wer in C programmiert, muß wissen, was er tut".

Es gibt keine mehrdimensionalen Vektoren, sondern nur „Vektoren von Vektoren", was im Prinzip auf dasselbe hinauskommt. Ein Schachbrett, das bekanntlich aus 8*8 Feldern besteht, könnte man so darstellen:

```
char Brett[8][8];
```

Entsprechend kann man dann mit „Brett[i][j]" auf das Element in der „i"-ten Zeile und „j"-ten Spalte zugreifen, oder umgekehrt. Das hängt ganz davon ab, ob Sie den ersten Index nun als Zeilen- oder Spaltennummer betrachten. Andererseits stellt dann aber auch „Brett[i]" eine einzelne Zeile (bzw. Spalte) dar. Wir haben es also tatsächlich mit einem Vektor von Vektoren und nicht einem einzelnen zweidimensionalen Vektor zu tun.

2.1.2.2 Zeiger und Vektoren

Jetzt werden Sie sich vielleicht wundern, aber ein Vektor und ein Zeiger sind in C fast dasselbe! Dies gehört zu jenen komischen Features, die den Leuten damals bei der Einführung von C ungeheuer toll vorkamen, heute aber nur noch ärgerliche Anachronismen sind.

Die Sache ist, kurz gesagt, folgende: Ein Pointer verweist normalerweise auf eine Speicherstelle. Aber oft sind ja auch die Speicherbereiche davor und dahinter von Interesse. Deshalb kann man einen Zeiger ebenfalls mit einem Index versehen, als ob er ein Vektor wäre. Betrachten wir einmal folgende Deklarationen:

```
int vector[100];
int *pointer = &vector[20];
```

Nun zeigt „pointer" auf das zwanzigste Element von „vector", und wir können „*pointer" synonym zu „vector[20]" benutzen. Intern wird ein Vektor aber realisiert, indem die Elemente hintereinander im Speicher liegen. Direkt hinter „vector[0]" liegt im Speicher also „vector[1]", worauf „vector[2]" folgt usw. Jetzt machen wir eine kleine Wertzuweisung, die Sie vielleicht verwundern wird:

```
pointer = pointer+5;
```

Wird zu einem Zeigerausdruck ein ganzzahliger Ausdruck addiert, so ist das Ergebnis ein Zeiger auf die um entsprechend viel Vektorelemente weiter liegende Speicheradresse. Also ist „pointer+5", wenn „pointer" auf das 20-ste Element von „vector" zeigt, ein Pointer auf „vector[25]". Man kann im Prinzip zu jedem Zeigerausdruck etwas addieren, aber Sinn hat das natürlich nur, wenn der Zeiger gerade in einen Vector zeigt.

Entsprechend kann man auch die Subtraktion „-“ und die Operatoren „+=“, „-=“, „++“ und „--“ auf Zeiger anwenden: Nach

```
pointer -= 5;
```

zeigt unser „pointer“ wieder auf das zwanzigste Element von „vector“.

Nun dürfte klar sein, daß

```
*(pointer+n)
```

den Inhalt des Vektorelements liefert, der „n“ Positionen hinter dem liegt, auf das „pointer“ gerade zeigt. Und dafür gibt es (die Überraschungen reißen nicht ab) eine schönere Schreibweise:

```
pointer[n]
```

Das sieht jetzt aus wie ein ganz normaler Vektorzugriff, und das ist es auch. Wieso das?

Computer können von Natur aus leicht mit Zeigern, die ja im Prinzip nur eine Speicheradresse repräsentieren, umgehen. Mit so großen und unhandlichen Gebilden wie Vektoren tun sie sich dagegen relativ schwer, was ganz einfach daher kommt, daß ein so großes Datenobjekt nicht in ein Prozessorregister paßt, sondern langsam und umständlich durch den Speicher geschoben werden muß. Also entschied man bei der Definition von C, Vektoren stets nur als Zeiger zu verwenden: Nach der Deklaration

```
double dat[10];
```

wird für „dat“ zwar ein zehn Elemente umfassender „double“-Vektor eingerichtet. Der Bezeichner „dat“ ist dann aber ein Ausdruck des Typs „Zeiger auf double“ und entspricht einem Zeiger auf das erste (also „nullte“) Vektorelement:

```
dat    ist identisch mit    &dat[0]
```

Folglich könnte man schreiben:

```
int vector[100];
int *pointer = vector;  // Entspricht "&vector[0]"
```

Um „pointer“ wieder auf das zwanzigste Element zu setzen, kann man also

```
int vector[100];
int *pointer = vector+20;
```

deklarieren. Der „normale“ Vektorzugriff

```
v[i]
```

ist nur eine andere Schreibweise für

```
*(v+i)
```

Mit diesen Features kann der echte C-Freak zwar echt lässig durch die Gegend pointern, aber übersichtlich ist das natürlich nicht.

An dieser Stelle ist wohl wieder ein Beispiel angebracht: Das folgende Beispiel setzt alle Elemente eines „int“-Vektors auf Null, benutzt dafür aber keine Schleife von 0 bis 99, sondern läßt einen Zeiger vom 0-ten bis zum 99-sten Element laufen:

```
void main()
{ int vector[100];
  for(int *pointer = vector; pointer != &vector[100];
    pointer++)
    *pointer = 0;
}
```

Der Zeiger „pointer“ wird also zu Beginn der „for“-Schleife auf das erste Vektorelement gesetzt. Solange „pointer“ nicht auf das hundertste Element zeigt (das es eigentlich gar nicht gibt, da der Vektor nur bis 99 reicht), wird das von „pointer“ referierte Element auf 0 gesetzt und anschließend der Zeiger inkrementiert, also um ein Element weitergesetzt.

Nervenstarken C-Anfängern sei noch gesagt, daß ein ultimativer Hard-Core-C-Coder folgendes geschrieben hätte:

```
void main()
{ int vector[100], *pointer = vector+100;
  while(pointer > vector)
    *--pointer = 0;
}

/* Der Zeiger "pointer" wird mit der Endadresse des Vektors
"vector" initialisiert. Solange er größer ist als die
Anfangsadresse von "vector", wird er dekrementiert, d. h. um
ein Vektorelement zurückgesetzt, und dann das referierte
Vektorelement auf 0 gesetzt */
```

Einer der vielen Schwachpunkte des Vektorkonzepts in C besteht darin, daß man Vektoren nicht einander zuweisen kann. Folgendes ist z. B. nicht möglich:

```
void main()
{ int v1[50], v2[50];  // Zwei Vektoren

  v1 = v2;             // ERROR
}
```

Das folgt direkt aus diesem seltsamen Vektorkonzept, denn „v1" ist demzufolge ja nur die Anfangsadresse der ersten Vektorelements, und eine Adresse allein ist kein L-Wert. Somit wäre die Zeile

```
v1 = v2; // Gedacht als Wertzuweisung zwischen Vektoren
```

gleichbedeutend mit

```
&v1[0] = &v2[0]; // Gleichsetzen von Adressen? Häh???
```

und das hat ja nun wirklich keinen Sinn. Wenn man in C einen Vektor in einen anderen kopieren will, muß man das (am besten mit einer Schleife) elementweise tun.

2.1.2.3 Genaueres über Vektordeklarationen

Bisher haben wir Vektoren immer in der Form

```
Grundtyp Variablenname [ Anzahl ]
```

deklariert. Eine gültige Typbeschreibung erhält man in C immer, indem man bei einer Variablendeklaration den Variablennamen wegläßt. Also ist

```
int [10]
```

die korrekte Beschreibung des Datentyps „int-Vektor mit zehn Elementen".

Natürlich kann man Zeiger- und Vektortypen miteinander verbinden. Dann kann es aber zu etwas kryptischen Deklarationen kommen, z. B.

```
char *komisch[20];
```

Was ist das jetzt, ein Vektor von Zeigern oder ein Zeiger auf einen Vektor? Hier helfen wieder die beiden goldenen Regeln für das Überleben verworrener C-Deklarationen:

1. Typen und Deklarationen liest man von innen nach außen.
2. Die Deklaration eines Objekts soll idiotischerweise wie die Benutzung aussehen.

Nach Regel 2 könnten wir obiges Objekt beispielsweise so benutzen:

```
*komisch[i]
```

(wobei das Ergebnis dann, wie die Deklaration ja sagt, ein „char" ist - wenigstens das wissen wir ja). Nach Regel 1 fangen wir bei der Interpretation innen (beim Bezeichner „komisch") an und müssen nur noch herausfinden, ob wir jetzt zuerst den Index oder das „*" anwenden. Wenn Sie gut aufgepaßt haben, wissen Sie noch, daß ein Index als Postfix-Operator gilt, daß „*" unter die Rubrik „unärer Operator" fällt und daß Postfix-Operatoren stärker binden als unäre Operatoren. Folglich wird zuerst „[i]" auf „komisch" angewendet und erst dann „*" auf das Ergebnis. Ergo ist „komisch" ein Vektor von Zeigern auf „char".

Um einen Zeiger auf einen Vektor zu deklarieren, müssen Sie also Klammern setzen:

```
char (*sehrkomisch)[20];
```

Dann liefert

```
(*sehrkomisch)[i]
```

das „i"-te Zeichen in dem „char"-Vektor, auf den „sehrkomisch" zeigt. Ganz schön komisch, was?

So, und spätestens jetzt dürfte Ihnen wohl klar sein, daß es in der C-Syntax immer noch etwas schlimmer kommen kann, als man denkt. Also dürfte es Sie auch nicht weiter schockieren, daß man die Kombination von Zeiger- und Vektortypen beliebig weit treiben kann, etwa

```
int *(*(**x)[2])[3]
```

Das ist dann - immer schön nach unseren beiden Regeln - ein Zeiger auf einen Zeiger auf einen Vektor zweier Zeiger auf je einen dreielementigen Vektor von Zeigern auf int. Das einzig Schöne an solchen Datentypen ist, daß man sie normalerweise nicht braucht und sich deshalb nicht darum kümmern muß.

Dagegen tauchen Datentypen wie

```
int (*)[10]
```

tatsächlich bisweilen auf (z. B. in Parameterlisten) und bringen auch erfahrene C-Programmierer in Verlegenheit. Die Klammern um das Pointersymbol „*“ sehen doch ziemlich seltsam aus, zumal

```
int *[10]
```

etwas ganz anderes ist! Hier sollten Sie immer daran denken, daß Datentypen in C dummerweise wie Variablendeklarationen aussehen, nur daß man eben den Variablennamen wegläßt, und der müßte hier in beiden Fällen direkt hinter dem „*“ stehen. Also ist ersteres die korrekte Schreibweise für „Zeiger auf Vektor“, während letzteres „Vektor von Zeigern“ heißt.

2.1.2.4 Vektoren als Parameter

Oft will man einen Vektor als Argument an eine Funktion übergeben. Leider kann C auch das nicht, jedenfalls nicht so richtig: das geht nur über Pointer. Nehmen wir einmal an, Sie deklarieren folgende Funktion:

```
void Initialisiere (int v[])
{ for (int i=0; i<10; i++)
    v[i] = 0;
}

void main()
{ int vector[10];
  Initialisiere(vector);
}
```

Wenn man einen skalaren Datentypen - also numerisch oder Pointer - als Argument übergibt, wird bekanntlich eine Kopie des Argumentwerts auf dem Stack abgelegt. Bei Vektor-Parametern macht C aber Probleme: Bei der Funktionsdeklaration ersetzt der Compiler automatisch „int[10]“ durch „int*“. Der Parameter „v“ ist dadurch also kein Vektor, sondern bloß ein Zeiger auf einen solchen. Es hätte auf das Programm keinerlei Einfluß gehabt, wenn man

```
void Initialisiere (int *v)
```

deklariert hätte.

Daran erkennen Sie unschwer, daß es keine rechte Bedeutung hat, ob wir als Argument nun einen Vektor mit 10 oder 100 Elementen übergeben. Deshalb ist bei Parameterdeklarationen auch die folgende Schreibweise erlaubt:

```
void Initialisiere (int v[])
```

Hier sieht man dann sofort, daß prinzipiell ein beliebiger „int"-Vektor als Argument verwendet werden könnte. Wenn der dann allerdings weniger als zehn Elemente hat, schreibt die Funktion gnadenlos darüber hinaus und bringt den Recher schätzungsweise zum Absturz. Deshalb ist es in solchen Situationen klüger, die Vektorgröße als zusätzlichen Parameter zu erwarten, etwa so:

```
void Initialisiere (int v[], int n)
{ for (int i=0; i<n; i++)
    v[i] = 0;
}

void main()
{ int vector[10];
  Initialisiere(vector, 10);
}
```

Mehrdimensionale Vektoren gibt es nicht, nur Vektoren von Vektoren. Erwartet eine Funktion ein derartiges Argument, etwa

```
void Ausgabe (char c[10][80]);
```

so wird in Wirklichkeit ein Zeiger auf „char[80]" übergeben. Also wären folgende Deklarationen äquivalent:

```
void Ausgabe (char c[10][80]);
void Ausgabe (char c[ ][80]);
void Ausgabe (char (*c)[80]);
```

Syntaktisch falsch sind dagegen folgende Deklarationen:

```
void Ausgabe (char c[ ][ ]);
void Ausgabe (char c[10][ ]);
```

In beiden Fällen weiß der Compiler zwar, daß der Parameter ein Zeiger auf einen Vektor ist, aber nicht, wie viele Elemente dieser Vektor hat. Also kann er damit nichts Rechtes anfangen, und er beschwert sich.

2.1.2.5 Strings

2.1.2.5.1 Konstante Zeichenketten

Bisher hatten wir Zeichenketten nur in konstanter Form kennengelernt. Ein konstanter String ist aber nichts anderes als ein Vektor von Zeichen. Damit man das Ende einer Zeichenkette feststellen kann, wenn man ihre Länge gerade nicht kennt, steht in C am

Ende einer Zeichenkette immer ein Nullbyte, wodurch ein String immer um ein Element verlängert wird. Deshalb hat der Ausdruck

```
"Hallo"
```

den Typ

```
char[6]
```

Man kann eine konstante Zeichenkette sogar mit einem Index versehen:

```
cout « "Hallo"[1];
```

```
gibt a aus.
```

Eine konstante Zeichenkette kann an einen „char"-Zeiger zugewiesen werden, denn allgemein sind in C Zuweisungen „T* = T[ ]" für beliebige Typen „T" erlaubt. Ein Beispiel:

```
char *string1;
```

```
string1 = "Mach's gut, und danke für den Fisch!";
```

Natürlich **zeigt** „string1" jetzt auf die entsprechende konstante Zeichenkette und enthält nicht etwa eine Kopie davon. Deshalb ist es zwar erlaubt, aber eine ziemlich haarige Angelegenheit, eine Stringkonstante zu verändern. Auch dazu ein Beispiel:

```
char *st1 = "Hallo"; st1[1] = 'e';
```

Nun sollte die konstante Zeichenkette von „Hallo" in „Hello" verwandelt worden sein, was natürlich ziemlich komisch ist, denn auf diese Art und Weise ist eine konstante Zeichenkette ja alles andere als konstant. Deshalb wird auch allgemein von solchen Konstruktionen abgeraten. Noch dicker kommt es, wenn der Compiler Zeichenketten optimiert:

```
char *st1 = "Test", *st2 = "Test";
st1[2] = 'x';   // Ändere st1 von "Test" nach "Text"
```

Zunächst zeigen beide Pointer auf eine Konstante namens „Test", aber der ANSI-Standard legt nicht fest, ob es sich dabei um dieselbe Konstante handelt! MaxonC++ legt hier zwei Kopien des Strings an, also zeigt „st2" nachher immer noch auf „Test". Ein anderer Compiler könnte aber durchaus nur einmal die Konstante „Test" anlegen, so daß „st1" und „st2" auf ein und denselben Speicherbereich zeigen, mit der Folge, daß über „st1" immer auch „st2" modifiziert wird. Also Pfoten weg von Schreibzugriffen auf konstante Zeichenketten!

2.1.2.5.2 Initialisieren, Kopieren und Zuweisen

Ein Vektor von „char"-Elementen kann mit einer konstanten Zeichenkette initialisiert werden:

```
char c[20] = "Hallo";
```

Nach dieser Initialisierung gilt:

```
c[0] == 'H', c[1] == 'a', ..., c[4] == 'o', c[5] == '\0'
```

Die restlichen Elemente von „c" sind undefiniert.

Bei einer solchen Initialisierung muß der String einschließlich des Nullbytes am Ende in die Variable hineinpassen, sonst meldet der Compiler einen Fehler. Dabei gibt es wieder einen jener spitzfindigen Unterschiede zwischen C und C++: In ANSI C wird bei solchen Initialisierungen nicht verlangt, daß das Nullbyte in die Zeichenkette passen muß. Also ist folgendes unter MaxonC++ nur im C-Modus möglich:

```
#pragma -
char s[4] = "Test";  // 4 Zeichen + Nullbyte: nur in C erlaubt!
```

Wenn man diesen String dann aber ausgeben läßt, kann es sein, daß man eine Überraschung erlebt, denn dabei wird stets alles vom Stringanfang bis zum ersten Nullbyte ausgegeben. Da „s" selbst hier kein solches Endzeichen besitzt, wird auch noch das ausgegeben, was zufällig gerade im Speicher hinter „s" liegt.

Strings sind elementare, nahezu unverzichtbare Datentypen. Daher ist es schade, daß C sie kaum unterstützt. Wertzuweisungen zwischen „char"-Vektoren sind, genau wie bei allen anderen Vektoren, nicht direkt möglich:

```
#include <stream.h>

void main()
{ char str1[20] = "Herforder Pils", str2[20];
  str2 = str1;    // ERROR
  cout « str2;
}
```

Hier bleibt uns nur eins übrig: Wir müssen die Strings Zeichen für Zeichen kopieren. Damit das nicht zu mühsam wird, schreiben wir dazu eine kleine Funktion namens „strcpy" wie „String-Copy". Erfreulicherweise wird ein String immer mit einem Nullzeichen abgeschlossen. Deshalb braucht „strcpy" nicht zu wissen, wie lang der zu kopierende String ist, sondern kann das selbst feststellen. Auf diese Weise kann man dann „str2 = str1" einfach durch

```
strcpy(str2, str1)
```

ersetzen.

Ein erster Entwurf von „strcpy" könnte so aussehen:

```
void strcpy(char ziel[ ], char quell[ ])
{ int i;
  for (i=0; quell[i] != '\0'; ++i) // Zeichenweise von "quell"
  ziel[i] = quell[i];  // nach "ziel" kopieren
  ziel[i] = '\0';      // Nullbyte ans Ende von "ziel"
}
```

Das ist eine übersichtliche, gut lesbare Version von „strcpy", wie sie ein C-Anfänger schreiben würde. Ein Fortgeschrittener würde dagegen folgendes tun:

1. Man deklariert die Parameter gleich als Pointer, denn der Compiler wandelt Vektor-Parameter ja sowieso dahin um.
2. Man kann „quell[i]" direkt als Bedingung verwenden, denn das bedeutet ja nichts anderes als „quell[i] != 0".
3. Wenn man direkt mit Pointern operiert, spart man sich die Indexvariable.
4. Die Konstante „'\0'" kann man ganz einfach durch „0" ersetzen, denn in C sind Zeichen ja nichts anderes als Zahlen.

Eine fortgeschrittenere Version von „strcpy" sähe dann etwa so aus:

```
void strcpy(char *ziel, char *quell)
{ while (*quell) // Solange „quell" auf ein Zeichen != 0 zeigt, ...
  *ziel++ = *quell++;  // wird dieses nach „*ziel" kopiert,
                       // und beide Zeiger werden
             // um je ein Zeichen weitergesetzt.
  *ziel = 0; // Nullbyte ans Ende des Zielstrings schreiben.
}
```

Bitte erzählen Sie mir jetzt nicht, daß das ganz schön unübersichtlich ist - ich bin mir darüber im klaren. Aber erstens ist diese Version kürzer und zweitens schneller als die alte.

In der Praxis brauchen Sie sich zum Glück nicht den Kopf zu zerbrechen, wie Sie „strcpy" nun implementieren, denn diese Funktion gibt es bereits fix und fertig. Sie wird im Includefile „<string.h>" deklariert und bei Bedarf vom Linker eingebunden:

```
#include <stream.h>
#include <string.h>

void main()
{ char str1[20] = "Herforder Pils", str2[20];
  strcpy (str2, str1);
  cout « str2;
}
```

Bitte denken Sie daran, daß „strcpy" eine Zeichenkette gandenlos kopiert, ganz egal, ob sie nun in die Zielvariable hineinpaßt oder nicht. Unter Umständen wird dann über die Stringvariable hinausgeschrieben, und das hat meist fatale Folgen. Etwas sicherer ist deshalb die Funktion „strncpy", die als zusätzliches Argument die maximale Anzahl der zu kopierenden Strings erhält. Dabei wird aber nicht automatisch ein Nullbyte an das Stringende gesetzt, sondern nur dann, wenn der zu kopierende String auch tatsächlich in den Zielbereich hineinpaßt.

Ein Beispiel dafür:

```
#include <stream.h>
#include <string.h>

void main()
{ char s1[40] = "Live fast - love hard - die young.",
                                                  // J. Joplin
  s2[40] = "On the sunny side of the street.";  // Cadillac

  strncpy(s2, s1, 12);  // maximal 12 Zeichen kopieren

  cout « s2;       // Ergebnis: "Live fast -  side of the street."
}
```

Diesen Effekt kann man auch ausnutzen, um gezielt in einen String hinein zu kopieren. Auch dazu ein Beispiel:

```
#include <stream.h>
#include <string.h>

void main()
{ char s1[20] = "Piss off!";

  strncpy(s1+1, "***", 3);    // String wird zensiert

  cout « s1;       // Ausgabe: „P*** off!"
}
```

Hätte man hier „strcpy“ statt „strncpy“ benutzt, wäre auch das vierte Zeichen der Konstanten „***“, nämlich das Nullbyte, kopiert und so „s1“ dahinter abgeschnitten worden.

2.1.2.5.3 Stringvergleiche

Oft will man den Inhalt von Strings vergleichen, etwa um festzustellen, ob sie gleich sind oder ob ein String alphabetisch vor dem anderen steht. Dann geht das Elend mit den C-Strings schon wieder los: Wenn man Strings, also Vektoren von Zeichen, mit den normalen Operatoren wie „==“ oder „<„ vergleicht, vergleicht man effektiv Zeiger auf Zeichen und somit die Anfangsadressen der Strings. Ein Beispiel:

```
void main()
{ char s1[10] = "Test",
  s2[10] = "Test",
  *s3 = s1,
  *s4 = "Test";        // Es gilt: s1 == s3, aber s1 != s2 und s1 != s4
}
```

Hier wird die Zeichenkette „Test“ in die Stringvariablen „s1“ und „s2“ kopiert, und „s4“ zeigt auf eine weitere Kopie dieses Strings, während „s3“ auf den String „s1“ verweist. Obwohl alle drei Variablen bei der Ausgabe (z. B. „cout « s1“;) „Test“ liefern, sind nur „s1“ und „s3“ gleich (weil die Adresse übereinstimmt), während alle anderen Variablen voneinander verschieden sind.

Derartige Vergleiche sind manchmal wirklich sinnvoll, aber meist will man den tatsächlichen Inhalt von Strings vergleichen und nicht ihre Speicheradressen. Auch hier hilft uns eine Bibliotheksfunktion aus „<string.h>“ aus der Klemme: Sie heißt „strcmp“ und erhält als Parameter zwei Zeichenketten (das heißt in C immer: Zeiger darauf).

„strcmp(a,b)“ ist: = 0, wenn „a“ und „b“ gleich sind
< 0, wenn „a“ im Alphabet vor „b“ steht
> 0, wenn „a“ nach „b“ kommt.

„Alphabetische Reihenfolge“ bezieht sich hier auf den ASCII-Code. Es werden zunächst die jeweils ersten Zeichen der Strings verglichen. Sind diese gleich, werden die jeweils folgenden Zeichen verglichen und so weiter. Die Strings werden höchstens bis zum Nullbyte am Ende verglichen. Dieses Verfahren liefert im allgemeinen nicht die Reihenfolge, die man sich so landläufig unter „alphabetisch“ vorstellt. Insbesondere kommen für den Computer alle Großbuchstaben vor sämtlichen Kleinbuchstaben.

Mit

```
if (strcmp(x,y) == 0)
```

kann man also feststellen, ob der Inhalt der Zeichenketten „x“ und „y“ gleich ist. Natürlich kann man das auch verkürzen zu

```
if (!strcmp(x,y))
```

Um ein bißchen den Umgang mit Strings zu üben, wollen wir hier die Funktion „strcmp“ einmal selbst schreiben. Ein erster, simpler Versuch wäre

```
int strcpy(unsigned char s1[ ], unsigned char s2[ ])
{ int i = 0;

  // Wir übergehen alle identischen Zeichen
  // am Anfang der Strings:
  while (s1[i] != '\0' && s2[i] != '\0' && s1[i] == s2[i])
    i++;

  // Ergebnis auswerten:
  if (s1[i] < s2[i])
    return -1;
  else
    if (s1[i] > s2[i])
      return +1;
    else
      return 0;
}
```

Bei „strcpy(„Test“, „Text“)“ wird hier „i“ zweimal heraufgezählt, denn die beiden ersten Zeichen der beiden Zeichenketten sind ja identisch. Dann werden die Zeichen ‘s’ und ‘x’ verglichen, und als Ergebnis wird „-1“ zurückgegeben, denn ‘s’ ist kleiner als ‘x’. Hilfreich ist hier, daß das Nullzeichen, das ja das Stringende markiert, bei vorzeichenlosen „char“’s kleiner als alle anderen Zeichen ist. Deshalb kann man, wenn man mit „unsigned char“-Werten arbeitet, relativ problemlos den Fall, daß bei einem String das Stringende erreicht wird, abhandeln. Beim Vergleich von „Test“ mit „Teststring“ werden vier Zeichen übersprungen und dann das Nullzeichen am Ende von „Test“ mit dem zweiten ‘s’ aus „Teststring“ verglichen. Das Nullbyte ist natürlich kleiner als das ‘s’ und also „ Test“ kleiner als „Teststring“, wie man das auch nicht anders erwartet.

Optimal (im Sinne eines „harten“ C-Programmierers) ist die Funktion aber noch lange nicht. Damit uns niemand „Lamer“ nennt, ändern wir umgehend einige Details der Funktion:

1. Die Parametertypen ändern wir in „unsigned char*“ um, denn der Compiler tut das ja sowieso. Außerdem lassen wir das „unsigned“ wegfallen und sehen später, wie wir damit zurecht kommen.

2. Wenn wir schon mit Zeigern arbeiten, dann bitte auch richtig. Also fällt auch das „i“ flach, und wir setzen direkt die Zeiger weiter.

3. Niemand verlangt von uns, daß wir bei unterschiedlichen Strings gerade „-1“ oder „+1“ zurückgeben, es muß eben nur das Vorzeichen stimmen. Deshalb können wir die beiden Vergleichsoperationen durch eine simple Subtraktion ersetzen.

Mit diesen Änderungen sieht das Ganze so aus:

```
int strcpy(char *s1, char *s2) // Zweite Version
{ // Wir übergehen alle identischen Zeichen
  // am Anfang der Strings:
  while (*s1 != '\0' && *s2 != '\0' && *s1 == *s2)
  { ++s1;
    ++s2;
  }

  // Ergebnis auswerten:
  return (unsigned char)*s1 - (unsigned char)*s2;
}
```

Wir suchen also wieder das erste Zeichen, in dem sich „s1“ und „s2“ unterscheiden. Diese Zeichen wandeln wir dann in „unsigned char“ um und berechnen die Differenz. Das geschieht übrigens den Sprachregeln von ANSI C entsprechend im Typ „int“, so daß wir uns keine Sorgen wegen Überläufen machen müssen.

Die Funktion ist jetzt schon viel kürzer geworden, aber zwei Details stören immer noch: Zum einen sind die Vergleiche „!= '\0'“ redundant, da C ja nicht zwischen numerischen und logischen Datentypen unterscheidet, zum anderen kann man einen der beiden ersten Vergleiche weglassen, wie im folgenden geschehen:

```
int strcpy(char *s1, char *s2) // Dritte Version
{ while (*s1 && *s1 == *s2)
  { ++s1;
    ++s2;
  }

  return (unsigned char)*s1 - (unsigned char)*s2;
}
```

Für alle, die daran zweifeln, daß man das „*s2 != '\0'“ wirklich weglassen kann, sei gesagt, daß es vier verschiedene Fälle gibt:

- Die Strings sind gleich. Dann erreichen wir mit beiden Zeigern gleichzeitig das Stringende und brechen die Schleife aufgrund der Klausel „*s1“ ab.

- Die Strings unterscheiden sich schon vor beiden Stringenden. Dann stoßen wir irgendwann auf eine Stelle, wo „*s1 == *s2“ nicht mehr erfüllt ist, und die Schleife endet wiederum wunschgemäß.

- Wir erreichen zuerst das Ende von „s1“, da beide Strings bis hierhin identisch sind. Dann ist „*s1“ falsch, und die Schleife bricht ab.

- Wir erreichen zuerst das Ende von „s2“. Dann ist zwar „*s1“ noch erfüllt, aber „*s1 == *s2“ gilt natürlich nicht mehr, denn s2 zeigt auf ein Nullbyte und s1 nicht.

Die dritte Version von „strcmp“ ist wieder einmal ein schönes Beispiel dafür, wie man in C äußerst kurze und schnelle Programme schreiben kann, die dann aber leider niemand mehr ohne weiteres versteht.

Am Ende dieses Abschnitts sei noch die Funktion „strlen“ erwähnt, die ebenfalls in „<string.h>“ deklariert wird. Sie liefert die Länge einer Zeichenkette, und zwar ausschließlich des Nullbytes am Ende. Spaßeshalber versuchen wir auch hier eine eigene Implementierung. Zuerst wieder eine einfache, verständliche Version:

```
int strlen(char *s)  // Erste Version
{ int i = 0;
  while (s[i] != '\0')
    ++i;
  return i;
}
```

Das optimieren wir wieder, indem wir von Index- auf Zeigeroperationen umsteigen. Wir müssen uns dabei den Stringanfang merken und einen zweiten Zeiger bis zum Ende wandern lassen. Am Ende subtrahieren wir dann die beiden Pointer und erhalten so die Stringlänge (Sie erinnern sich: Die Differenz zweier Zeiger gleichen Typs ist die Anzahl der Vektorelemente, die dazwischen paßt):

```
int strlen(char *s)  // Zweite Version
{ char *t = s;
  while (*t)
    ++t;
  return t-s;
}
```

Das ist dann in der Tat eine Funktion, die sich wohl kaum weiter optimieren läßt - es sei denn, Sie verwenden die Bibliotheksfunktion „strlen“ von MaxonC++, denn die ist in Assembler geschrieben und deshalb noch etwas kürzer und schneller.

2.1.2.5.4 Argumente aus der Kommandozeile

Weil Sie gerade Strings kennengelernt haben und es sonst nirgendwo sinnvoll in dieses Handbuch paßt, verrate ich Ihnen hier, was es mit „argc" und „argv" auf sich hat:

Wenn man auf dem Amiga ein Programm von CLI startet, kann man hinter dem Programmnamen bekanntlich noch Argumente angeben. Natürlich haben diese Argumente meist einen tieferen Sinn, und folglich muß das Programm, das z. B. in C++ geschrieben ist, diese auch auswerten.

Für diesen Zweck gibt es in C eine standardisierte Lösung, die Argumente werden als Parameter an das Hauptprogramm, also „main", übergeben. Wenn man sie haben will, muß man folgende Parameter für „main" deklarieren:

```
void main(int argc, char *argv[ ]);
```

Der erste Parameter, der in der Regel „argc" genannt wird, enthält die Anzahl der Argumente, der zweite ist ein Vektor mit Zeigern auf eben diese, wobei der Programmname ebenfalls als Argument zählt. Vor dem Start des Programms wird nämlich die Argumentzeile in Worte zerlegt:

```
#include <stream.h>

void main(int argc, char *argv[])
{ int i;
  for (i=0; i<argc; ++i)
    cout « "Parameter " « i « ": " « argv[i] « "\n";
}
```

Angenommen, Sie speichern das ausführbare Programm unter dem Namen „test" ab und geben im CLI

```
test Wer bin ich.
```

ein. Dann hat der Parameter „argc" den Wert 4, und die vier ersten Zeiger in „argv" zeigen auf die Strings „test", „Wer", „bin" und „ich.". Natürlich müssen Sie aus der integrierten Umgebung von MaxonC++ nicht extra ein Exe-File abspeichern, sondern können die Parameterübergabe mit dem Menüpunkt „Parameter" aus dem Menü „Optionen" simulieren. Als Programmname wird dabei natürlich nichts allzu Sinnvolles übergeben, den das Programm existiert ja nicht als ausführbare Programmdatei.

2.1.2.6 Initialisierung von Vektoren

Vektoren sind die ersten Datentypen, bei denen es einen Unterschied zwischen Initialisierung und Wertzuweisung gibt. Eine wirkliche Wertzuweisung ist nur elementweise möglich, aber initialisieren darf man sie. Sie haben bereits gesehen, daß man Stringvariablen mit einer konstanten Zeichenkette oder einem anderen Stringausdruck initialisieren darf, während man für eine entsprechende Wertzuweisung auf die „strcpy"-Funktion zurückgreifen müßte. Ganz allgemein sind für Vektoren zwei verschiedene Initialisierungen erlaubt:

1. Ein Vektorausdruck desselben Elementtyps. Die Anzahl der Elemente muß dabei nicht übereinstimmen. Hat der Ausdruck mehr Elemente als die Variable, werden die überzähligen ignoriert, und wenn es zu wenige sind, bleiben entsprechend viele Vektorelemente undefiniert.

Beispiel:

```
void main()
{ int v1[10];
  for (int i=0; i<10; i++)    // v1 wird initialisiert
    v1[i] = i+42;

  int v2[15] = v1; // initialisiert v2 mit v1, wobei v2[10]
                   // bis v2[14] undefiniert bleiben.
}
```

2. Die Anfangswerte der einzelnen Vektorelemente können auch in einer Liste aufgezählt werden:

```
short svec[10] = { 0, 1, 17, 4, 21, 26, 7, 31, 42, 4711 };
```

Gibt man zu viele Werte an, meldet der Compiler hier einen Fehler, während er es akzeptiert, wenn man weniger Elemente angibt. Diese werden dann mit Null initialisiert.

Mehrdimensionale Vektoren werden mit entsprechend geschachtelten Elementlisten initialisiert:

```
int vvi[4][2] = { {1, 2}, {2, 3}, {3, 4}, {4, 5} };
```

Hier ist „vvi" (Sie erinnern sich: Deklarationen immer von innen nach außen lesen!) ein vierelementiger Vektor von Paaren von int's, also muß er auch mit vier Zahlenpaaren und nicht etwa zwei Viertupeln initialisiert werden.

Alternativ kann man solche Schachtelungen auch „plattklopfen“:

```
int vvi[4][2] = { 1, 2, 2, 3, 3, 4, 4, 5 };
```

Der Effekt ist derselbe, aber natürlich ist das weniger schön.

Auch Zeichenketten kann man mit Elementlisten initialiseren, wenn man das unbedingt will:

```
char string[5] = { 'H', 'a', 'l', 'l', 'o' };
```

Bei so einer Initialisierung können Sie dann auch, wie hier geschehen, das sonst obligatorische Nulbyte am Stringende weglassen.

Ein recht angenehmes Feature von C ist (man beachte: Ich habe soeben erstmals ANSI-C gelobt!), daß man Vektorgrenzen durch die Initialisierung festlegen lassen kann. Bei der Deklaration

```
char wasndas[ ] = "Watt is'n das?";
```

wird die Stringvariable „wasndas“ gerade so groß gewählt, daß der String einschließlich Nullbyte am Ende hineinpaßt. Analog geht das auch mit Elementlisten:

```
int Daten[ ] = { 4, 2, 42 };
```

deklariert und initialisiert einen Vektor mit drei Elementen. Sogar mehrdimensional ist das möglich:

```
int irgendwas[ ][ ][ ] =
{{{1,2},{3}},{{17,4,66}},{{1,2,4}},{{99}}};
```

In der innersten Klammerebene, also auf dem dritten Indexlevel, werden hier jeweils ein, zwei oder drei Zahlen angegeben. Der Compiler nimmt dann das Maximum, also geht der dritte Indexbereich von 0 bis 2. Auf der mittleren Ebene werden, wie Sie durch genaues Hinsehen vielleicht erkennen können, ein bis zwei Elemente initialisiert, und auf der obersten Klammerebene werden vier Initialisierungsvektoren angegeben. Nicht initialisierte Daten werden wieder mit 0 beschrieben, also kommt effektiv

```
int irgendwas[4][2][3]  = {   { {   1,  2,  0}, {  3,  0,  0} },
                              { {  17,  4, 66}, {  0,  0,  0} },
                              { {   1,  2,  4}, {  0,  0,  0} },
                              { {  99,  0,  0}, {  0,  0,  0} }
                          };
```

dabei heraus.

Wenn man bei mehrdimensionalen Vektoren die Indexbereiche wegläßt, darf man die Initialisierungen natürlich nicht wieder „plattkloppen“: Bei

```
short falsch[ ][ ] = {17, 4, 21, 26, 7, 31, 42};
```

kann der Compiler ja nicht mehr herausfinden, wie groß die beiden Indexbereiche jeweils zu wählen sind. Eindeutig und deshalb erlaubt ist

```
short richtig[ ][2] = {17, 4, 21, 26, 7, 31, 42};
```

Im umgekehrten Fall,

```
short nichtrichtig[4][ ] = {17, 4, 21, 26, 7, 31, 42};
```

ist der Compiler aber nicht clever genug, die Elementanzahl durch 4 zu teilen und aufzurunden, und so ist auch hier das Ergebnis undefiniert.

Bei der hier mehrfach verwendeten Zahlenfolge handelt es sich übrigens um die Lottozahlen der nächsten Woche. Von etwaigen Gewinnen mit diesen Zahlen gehen 50 Prozent an mich.

Natürlich darf man auch Elementlisten- und Stringinitialisierung mischen:

```
char Namen[ ][ ] = { "DeGarmo", "Jackson", "Rockenfield",
"Tate", "Wilton" };
```

legt ein Feld von 5 Strings mit jeweils 12 Zeichen an, also gerade genug für unsere Daten. Das ist in der Tat ausgesprochen angenehm. Weniger angenehm ist, daß alles, was in diesem Kapitel beschrieben wurde, wirklich ausschließlich für Initialisierungen gilt und für normale Wertzuweisungen an Vektoren in C nicht erlaubt ist:

```
void main()
{
  int valsch[4];
  valsch = {0, 8, 15, 4711};    // SYNTAX ERROR
}
```

Na ja, man kann eben nicht alles haben.

2.1.3 Referenzen

2.1.3.1 Einführung

Schlafen Sie in letzter Zeit schlecht? Tauchen in Ihren Träumen ständig wüste Diagramme mit seltsamen Objekten und Zeigern darauf auf? Dann ist die Diagnose eindeutig: Sie haben die Sache mit den Pointern noch nicht so ganz verdaut. Aber Ihnen kann geholfen werden: C++ bietet Ihnen, im Gegensatz zu C, ein alternatives Datentypkonzept an, mit dem man oft Zeiger vermeiden kann - die Rede ist von Referenzen.

Intern ist ein Referenztyp nichts anderes als ein Zeigertyp, aber mit dem Unterschied, daß eine Referenz nicht wie ein Zeiger aussieht. Man deklariert Referenzen auch genau wie Zeiger, nur mit einem „&" statt „*":

```
int &ir;
```

Damit haben wir aber auch schon den ersten Fehler gemacht: Eine Referenz muß immer initialisiert werden. Der Compiler achtet also lobenswerterweise darauf, daß eine Referenz nicht ins Irgendwo, sondern auf ein real existierendes Datenobjekt verweist. Als Initialisierung wird normalerweise ein L-Wert des referierten Typs erwartet:

```
int i = 26731, j = 3718847;
int &ir = i;
```

Im weiteren Programmverlauf zeigt „ir" dann unveränderbar auf die Variable „i" und kann deshalb synonym zu „i" benutzt werden:

```
ir = 42;
```

ändert über „ir" den Wert von „i", und auch

```
ir = j;
```

setzt nicht etwa die Referenz „ir" auf „j" um, sondern ändert ebenfalls den Wert von „i", so daß es bei Referenzen einen grundsätzlichen Unterschied zwischen Initialisierung und Wertzuweisung gibt. Nach der Initialisierung verweist eine Referenz also immer auf dasselbe Datenobjekt, und es gibt keine Möglichkeit, das zu ändern. Sogar wenn man die Adresse einer Referenz nimmt, etwa „&ir", erhält man wieder nur die Adresse des Objekts, auf das „ir" zeigt, und nicht etwa die Speicherstelle, an der intern dieser Zeiger abgespeichert wird.

2.1.3.2 „VAR-Parameter" und temporäre Objekte

An dieser Stelle werden Sie natürlich fragen, was Sie denn davon haben, wenn Sie in einem Programm eine Variable über einen solchen Verweis ansprechen können. In der Form, in der Referenzen oben benutzt wurden, noch gar nichts.

Eine der nützlichsten Anwendungen für Referenztypen sind Funktionsparameter. Vielleicht erinnern Sie sich noch, daß bei der Argumentauswertung dieselben Regeln wie bei einer Initialisierung gelten. Wenn Sie also einen Parameter als Referenz auf einen Typen T deklarieren, wird beim Funktionsaufruf als Argument ein L-Wert des Typs T erwartet und die Referenz automatisch mit einem Verweis auf diese Variable initialisiert:

```
void halbiere (int &para)
{ para /= 2;
}

void main()
{ int i = 84;
  halbiere (i);
}
```

Die Funktion „halbiere" hat also einen Seiteneffekt auf die als Argument übergebene Variable. Pascal-Programmierer kennen dieses Feature als „VAR-Parameter", nur daß man in C++ Referenzen eben universell benutzen kann und nicht nur in Parameterlisten.

In diesem Zusammenhang gibt es ein ziemlich eigentümliches Feature, das im 1.0-Standard eher versehentlich enthalten war und vom 2.0-Standard nur noch „inoffiziell" unterstützt wird (d. h. es gehört offiziell nicht mehr dazu, aber es wird empfohlen, es trotzdem noch zu unterstützen, um die Kompatibilität der Programme zu erhalten): Der Ausdruck, mit dem eine Referenz initialisiert wird, muß weder ein L-Wert sein noch genau den richtigen Typ haben.

Beispielsweise könnte man die Funktion „halbiere" auch folgendermaßen aufrufen:

```
halbiere(42);
```

Was soll der Quatsch? Wird jetzt die Konstante „42" halbiert (was bekanntlich alle Probleme des Universums halbieren würde)? Nein, wir schließen hier Bekanntschaft mit einem Feature namens „temporäres Objekt". Der Compiler richtet hier ganz ohne Zutun des Programmierers eine namenlose „int"-Variable ein, initialisiert diese mit 42 und ruft „halbiere" dann auf dieser Phantom-Variablen auf. Das ist zwar noch ein recht nettes Feature, aber es wird ziemlich eklig, wenn beim Aufruf eine Typkonvertierung nötig wird:

```
long x = 12;
halbiere(x);
```

„x" ist zwar ein L-Wert, aber eben nicht vom Typ „int", und deshalb ist hier eine Typkonvertierung nötig, und das Ergebnis einer solchen Operation ist generell kein L-Wert, und zwar nicht einmal unter MaxonC++, wo die interne Darstellungen von „int" und „long" identisch sind (jeweils 32 Bit) und deshalb eigentlich keine wirkliche Operation ausgeführt werden muß. Jedenfalls generiert MaxonC++ auch hier ein temporäres Objekt, initialisiert es mit „x" und ruft die Funktion „halbiere" auf, die das temporäre Objekt halbiert - und der Programmierer wundert sich höchstwahrscheinlich, daß der Wert von „x" nicht halbiert wird (was im obigen Beispiel totsicher gemeint war).

Zu allem Überfluß funktioniert dieses Feature nicht nur bei Funktionsparametern, sondern bei sämtlichen Referenzen, z. B.

```
int &ir = 3.14159;     // Etwas komisch, aber erlaubt
char c;
signed char &cr = c;   // Überraschung!!!
```

Das Beispiel mit dem „ir" ist zwar ziemlich obskur. Hier wandelt der Compiler 3.14159 nach „int" um, initialisiert ein temporäres Objekt damit und läßt „ir" darauf verweisen, was natürlich ausgesprochen unnütz ist. Die Initialisierung von „cr" hingegen steckt voller Heimtücke. Da „char" und „signed char" in C++ zwei unterschiedliche Typen sind (obwohl auch sie in MaxonC++ praktisch identisch sind), verweist „cr" hier nicht etwa auf „c", sondern auf ein temporäres Objekt.

Im 2.0-Standard ist die Angelegenheit wesentlich eleganter gelöst. Ich muß an dieser Stelle etwas vorgreifen (nämlich auf Abschnitt 2.2.3) und schon jetzt verraten, daß man Typen aller Art mit „const" qualifizieren kann, etwa so:

```
void machwas(const double &D);
```

Die Referenz verweist hier auf ein „const double", was nichts anderes bedeutet, als daß man nicht beabsichtigt, den Wert jenes Datenobjekts zu verändern, wobei der Compiler das auch überwacht und jegliche Veränderung des referierten Objekts als Fehler meldet. Nun ist es ziemlich unerheblich, ob die Referenz ein direkter Zeiger auf das Funktionsargument oder nur ein Verweis auf ein temporäres Objekt ist, denn hier sind Überraschungen weitgehend ausgeschlossen: Da die Funktion das referierte Objekt nicht verändern kann, hat sie auch keine Seiteneffekte darauf, und der Programmierer wird nicht mehr mit der Tatsache konfrontiert, daß anstelle des Arguments unter Umständen bloß ein temporäres Objekt verändert wird.

Jedenfalls lautet die neue, korrekte Sprachregel so, daß eine Referenz auf ein nichtkonstantes Objekt mit einem L-Wert des richtigen Typs initialisiert werden muß, während bei Referenzen auf mit „const" qualifizierte Typen (und nur hier!) bei Bedarf temporäre Objekte eingerichtet werden.

2.1.3.3 Weitere Möglichkeiten und Einschränkungen

Zeiger auf Referenzen sind ebenso wenig erlaubt wie Vektoren von Referenzen:

```
int &*p1;      // ERROR, Zeiger auf Referenz
int &v1[10];   // ERROR, Vektor von Referenzen
```

Umgekehrt geht das natürlich:

```
int *&p2;      // Referenz auf Zeiger
int (&v2)[10]; // Referenz auf Vektor
```

Eine Funktion darf auch eine Referenz zurückgeben:

```
int &f(int i)
{ int j = 2*i;
  return j;
}
```

Diese Funktion ist zwar formal richtig, aber trotzdem falsch: Hier wird ein Verweis auf die lokale Variable „j" zurückgegeben, die nach Ausführung der Funktion nicht mehr existiert. Also zeigt die Referenz irgendwo in die Pampa, und eine Zuweisung an das Ergebnis der Funktion, etwa

```
int &f(int i); // Liefert wie oben Referenz auf ein "int"

void main()
{ f(1) = 42; // Ändert Wert des Objekts, auf das "f(1)" verweist
}
```

hätte unter Umständen fatale Folgen. So ganz sind Sie also auch bei Referenzen nicht gegen Pointer-Chaos gefeit.

Eine etwas sinnvollere Anwendung für Funktionen mit Referenz-Ergebnis wäre z. B. ein simulierter Vektor. Manchmal hat man eine Liste von Daten nicht als richtigen Vektor vorliegen, möchte sie aber trotzdem wie Vektorelemente benutzen. Das könnte so aussehen:

```
int i0, i1, i2, i3;   // Unsere vier Daten

int &Liste (int Index)
{ switch (Index)
  { case 0:
      return i0;
    case 1:
      return i1;
    case 2:
      return i2;
    case 3:
      return i3;
    default:  // ... Fehlermeldung ausgeben, Programm abbrechen
...
  }
}

void main()
{ int i;
  for (i=0; i<4; i++)
    Liste(i) = 42+i;

  Liste(0) = 2*Liste(1)+Liste(3);
}
```

Wie Sie sehen, können wir auf diese Weise unsere ganz normalen Variablen „i0" bis „i3" wie einen Vektor benutzen. Sinnvoll ist eine solche Konstruktion aber vor allem dann, wenn die fraglichen Daten nicht in Form von gewöhnlichen Variablen, sondern z. B. als dynamisch erzeugte verkettete Liste vorliegen - aber das gehört in ein anderes Kapitel. Vorher müssen wir noch unsere Hausaufgaben machen und uns durch diverse Features kämpfen, die uns C++ sonst noch so bietet.

2.2 Qualifizierungen, Deklarationen und Spezifikationen

2.2.1 Speicherklassen

2.2.1.1 Automatische und statische Daten

Ein C-Programm hat verschiedene Möglichkeiten, Daten abzuspreichern. Da wäre beispielsweise der Stack, ein RAM-Bereich, der für jeden Prozess eingerichtet wird und dann, wie der Name schon sagt, wie ein Stapel funktioniert: Man kann Daten drauflegen und sie später wieder herunternehmen.

In C-Programmen wird der Stack vor allem für lokale Daten von Funktionen und für ihre Argumente benutzt. Wenn eine Funktion benutzt wird, legt der aufrufende Programmteil die Argumente auf den Stack und ruft die Funktion als Unterprogramm auf, wobei der Prozessor dann übrigens auch noch die Rückkehradresse, an der er nach Beendigung der Funktion weitermachen soll, ebenfalls auf dem Stack vermerkt. Die Funktion richtet sich dann selbst genug Stackplatz für ihre lokalen Daten ein. Am Ende der Funktion, etwa bei einem „return"-Statement, werden die lokalen Daten wieder vom Stack entfernt und dann die Rückkehradresse heruntergenommen und angesprungen, so daß der Prozessor wieder im aufrufenden Programm landet. Diesem obliegt es dann, die Argumente wieder vom Stack zu entfernen.

Da C eine Hochsprache (oder zumindest als solche gedacht) ist, müssen Sie sich um solche Mechanismen natürlich nicht selbst kümmern. Wichtig ist nur, daß Ihnen die prinzipiellen Mechanismen, mit denen solche Variablen erzeugt und wieder zerstört werden, klar sind. In den Abschnitten 2.1.1.3 und 2.1.3 wurde ja bereits gesagt, was passieren kann, wenn man diese Interna mißachtet und Zeiger bzw. Referenzen auf lokale Daten aus einer Funktion exportiert.

Weil die Variablen auf dem Stack aus Sicht des C-Programmierers automatisch erzeugt und gelöscht werden, spricht man in diesem Zusammenhang auch von „automatischen" Variablen. Das Gegenteil davon sind „statische" Datenobjekte.

Statische Daten werden natürlich auch vollautomatisch eingerichtet, aber das nur einmal, nämlich am Programmanfang. Sie existieren dann bis zum Programmende. Globale Variablen, also solche, die außerhalb von Funktionen deklariert werden, sind grundsätzlich statisch.

Bei jeder Variablendeklaration (und bei einigen anderen Deklarationen auch) ist es möglich, die Speicherklasse explizit anzugeben. Dazu dienen hauptsächlich die Schlüsselwörter „auto" und „static":

```
int f()
{ auto int i, j;
  static int k;
}
```

Die beiden Variablen „i" und „j" werden jedesmal, wenn die Funktion „f" ausgeführt wird, neu eingerichtet und bei Beendigung der Funktion wieder vernichtet. Das Wortsymbol „auto" ist übrigens generell nutzlos und könnte ebenso gut ersatzlos gestrichen werden, und das macht man in der Praxis dann auch. Da keine Initialisierung angegeben ist, haben „i" und „j" zunächst undefinierte Werte.

Dadurch, daß der Deklaration von „k" das Schlüsselwort „static" vorangestellt wird, verhält diese Variable sich genau so, als wäre sie global, also auf Dateiebene, deklariert worden, mit einem wichtigen Unterschied: Sie ist auf diese Art und Weise ausschließlich innerhalb der Funktion „f" bekannt und benutzbar.

Alle statischen Daten werden beim Programmstart initialisiert, und zwar mit Null, sofern der Programmierer keine andere Initialisierung angibt. Wenn Sie beispielsweise am Anfang einer Funktion

```
static int count;
++count;
```

deklarieren, kann sie immer anhand von „count" feststellen, zum wievielten Mal sie aufgerufen wurde.

Das Symbol „auto" darf ausschließlich innerhalb von Funktionen verwendet werden, wo es bekanntlich überflüssig ist, da lokale Variablen ja sowieso schon „automatisch" sind, sofern nicht eine andere Speicherklasse spezifiziert wird. „static" kann im hier beschriebenen Sinne ebenfalls nur innerhalb von Funktionen benutzt werden. Auf Dateiebene, wo Variablen ja defaultmäßig statisch sind, hat es verwirrenderweise eine etwas andere Bedeutung, die im folgenden Abschnitt beschrieben wird.

ANSI C verlangt, daß statische Daten ausschließlich mit konstanten Werten initialisiert werden. C++ ist da weniger pingelig und läßt auch beliebige nicht-konstante Ausdrücke zu, die auch andere statische Variablen und sogar Funktionsaufrufen enthalten dürfen, beispielsweise

```
double f(int n);

int i = 0;
double d = f(i);     // C++, aber nicht C
long l = (long) d;   // Ebenfalls nur in C++
```

Undefinierte Ergebnisse erhält man dagegen, wenn man statische Variablen mit nichtstatischen Werten initialisiert, etwa

```
void f(int i)
{ static int j = i;  // Falsch!
}
```

Wie bereits gesagt, geschieht die Initialisierung statischer Daten am Programmbeginn, also zu einem Zeitpunkt, an der hier die Variable „i" noch nicht initialisiert ist, so daß „j" hier mit einem undefinierten Wert initialisiert wird.

2.2.1.2 Module und ihre Gültigkeitsbereiche

2.2.1.2.1 Gemeinsame Daten und Funktionen

Bisher habe ich Ihnen diskret verschwiegen, daß ein C-Programm ohne weiteres aus mehreren Modulen bestehen kann. Jedes Modul entsteht aus einem Quelltext (eventuell einschließlich Includedateien) und kann unabhängig von den anderen Modulen compiliert werden, weshalb man ein Modul in C etwas verschämt auch „Übersetzungseinheit" („translation unit") nennt. „Verschämt" deshalb, weil das Modulkonzept nicht besonders schön in die Programmiersprache C integriert wurde. Andere Hochsprachen, etwa Modula oder die allseits bekannten modernen Pascal-Dialekte, ermöglichen nämlich durch geeignete Sprachkonstrukte eine klare Trennung zwischen dem „Interface" eines Moduls, also dem Teil, der für andere Module sichtbar sein soll, und der „Implementierung", also den Programmteilen, die Privatsache des Moduls sein sollen. In C hat man ganz einfach festgelegt, daß jede Funktion und jedes auf Dateiebene deklarierte Datenobjekt öffentlich sind und von anderen Modulen benutzt werden können.

Besonders einfach ist das bei Funktionen: Wird eine Funktion deklariert, aber nicht definiert, weiß der Compiler, daß er sie vor irgendwo importieren muß. Beispielsweise könnte ein Modul folgendes enthalten:

```
// Datei "hello.c"

#include <stream.h>

void hello()
{ cout << "Hello, World!\n";
}
```

Dann braucht ein anderes Modul die Funktion nur noch zu deklarieren und kann sie dann benutzen:

```
// Modul "haupt"

void hello();

void main()
{ hello();
}
```

Natürlich muß man dem Linker irgendwie klarmachen, daß er das Programm aus den beiden Modulen „hello" und „haupt" zusammensetzen soll. In der Entwicklungsumgebung von MaxonC++ muß man dazu ein „Makefile" erzeugen, bei der Kommandozeilenversion reicht es, wenn man beide Quelldateinamen gleichzeitig als Parameter angibt. Näheres dazu steht im ersten Teil dieses Handbuchs.

Der Linker verlangt dann jedenfalls, daß irgend ein Modul eine Funktion „hello" exportiert. Es ist natürlich möglich, daß sie in mehreren Übersetzungseinheiten zugleich definiert wird. Dann meldet der Linker einen entsprechenden Fehler, weil er dann ja nicht weiß, welches „hello" er nun nehmen soll.

Bei gemeinsam benutzten Daten ist das Ganze etwas komplizierter, denn eine Deklaration wie „int i;" teilt dem Compiler bekanntlich nicht bloß mit, daß es eine Variable „i" gibt, sondern weist ihn auch an, eine solche zu erzeugen. Strenggenommen handelt es sich dabei also nicht um eine Deklaration, sondern eine Definition, aber der Begriff „Variablendeklaration" wird nun einmal traditionell benutzt. Jedenfalls hat jedes Modul, in dem „int i;" deklariert wird, sein eigenes „i", daß es zu allem Überfluß auch noch exportiert, so daß der Linker sich beklagen wird, daß dieses Symbol mehrfach definiert wird. So geht es also nicht.

Die Lösung dieses Problems ist die Speicherklasse „extern". Dadurch wird der Compiler angewiesen, ein Datenobjekt nicht zu erzeugen, sondern es zu importieren. „extern" wird syntaktisch genau wie „static" oder „auto" benutzt:

```
extern int i;
```

Man darf innerhalb einer Übersetzungseinheit jede Variable beliebig oft extern und zusätzlich höchstens einmal „normal" deklarieren, z. B.

```
extern int i;
int j;
extern int i, j;
int i;
```

Das erleichtert es enorm, die Schnittstelle zwischen Modulen in einem „Header-File" zu deklarieren. Normalerweise legt man zu jeder Übersetzungseinheit eine Includedatei an, in der alles deklariert wird, was vom jeweiligen Modul der „Allgemeinheit" zur Verfügung gestellt wird. Nehmen wir an, das Modul „hello.c" exportiere die Funktion „hello" und die Strungvariable „text". Das „Headerfile" dazu, das gewöhnlich „hello.h" oder „hello.i" genannt wird, enthält dann die Prototypen der exportierten Funktionen und „extern"-Deklarationen für die Variablen:

```
// * Header"hello.h" *
extern char text[];
void hello();
```

Das Modul „hello" definiert dann diese Namen. In die Schnittstellendeklaration gehören normalerweise nicht nur Variablen- und Funktionsdeklarationen, sondern auch Typdefintionen, die Gegenstand einiger folgender Kapitel sind. Deshalb muß im allgemeinen das Modul sein eigenes „Headerfile" einschließen, wobei der Dateiname allerdings in doppelten Hochkomma „..." und nicht wie bisher in spitzen Klammern <...> zu setzen ist, denn es handelt sich diesmal ja nicht um eine Standarddatei:

```
// * Modul "hello.c" *

#include "hello.h"   // Eigenes Headerfile
#include <stream.h>  // Wie gewohnt

char text[] = "Hello, World!";

void hello()
{ cout « text;
}
```

Das Hauptprogramm, nennen wir es „main.c", wird in der Regel nichts exportieren und braucht deshalb kein eigenes „Headerfile". Es benutzt lediglich das von „hello" und hat dann alle Deklarationen, die es braucht, um die Funktionen von „hello" zu benutzen:

```
// * Hauptmodul *

#include "hello.h"

void main()
{ char *txtptr = text;
  hello();
}
```

Übrigens darf man das Schlüsselwort „extern" auch vor Funktionsdeklarationen setzen, z. B.

```
extern void hello();
```

Das ist aber überflüssig, denn Funktionen, die nicht definiert werden, gelten ja automatisch als „extern".

2.2.1.2.2 Modulinterne „static"-Namen

Objekte und Funktionen, die auf Dateiebene deklariert werden, exportieren in der Regel ein Namenssymbol an den Linker, so daß alle anderen Module darauf zugreifen können. Manchmal führt das zu unerwarteten Effekten, wenn zwei Übersetzungseinheiten denselben Bezeichner für unterschiedliche modulinterne Zwecke definieren. Der Compiler stört sich daran nicht (denn er betrachtet ja immer nur eine Übersetzungseinheit, also ein Modul), aber der Linker mäkelt, weil ein Name mehrfach definiert wurde. Um derartige Kollisionen von vornherein zu verhindern, sieht C die Möglichkeit vor, Variablen und Funktionen als „nicht-exportiert" zu deklarieren. Aus nicht mehr nachvollziehbaren Gründen hat man sich entschieden, dafür das Schlüsselwort „static" zu überladen: Eine Deklaration wie

```
static char c;
```

hat schon in ANSI-C zwei Bedeutungen (und in C++ noch eine dritte, dazu an anderer Stelle mehr), je nach dem, wo sie steht:

- In einer Funktion deklariert sie eine Variable, die nicht auf dem Stack (im „automatischen" Speicher), sondern statisch im Datenbereich des Programms angelegt wird und dort die ganze Programmlaufzeit über existiert.

- Auf Dateiebene, also außerhalb aller Funktionsdefinitionen, deklariert sie ebenfalls eine Variable im statischen Speicher (denn alle Variablen auf Dateiebene sind statisch), deklariert aber zusätzlich, daß dieses Datenobjekt nicht anderen Modulen zugänglich sein soll. Es wird dafür insbesondere kein Symbol an den Linker weitergereicht, so daß es auch nicht zu versehentlichen Kollisionen kommen kann.

Bitte fragen Sie mich nicht, wieso man „static" für zwei ganz unterschiedliche Zwecke benutzt - ich weiß es nicht, und wenn ich etwas zu sagen hätte, würde man für letztere Funktion ein neues Schlüsselwort wie „intern", analog zu „extern", einführen, aber mich fragt ja keiner.

Jedenfalls ist man insofern konsistent, als daß man auch Funktionen als „static" deklarieren kann, etwa so:

```
static void hello();
```

Als kleine Erleichterung muß man das „static" nur bei der ersten Deklaration der Funktion hinschreiben, bei allen folgenden darf man es auch weglassen:

```
static void hello();

void hello()    // OK
{ // usw..
}
```

Umgekehrt ist es allerdings verboten:

```
void hello();

// ... irgendwas ...

static void hello(); // Error: "Inconsistent linkage"
```

Das liegt daran, daß der Compiler sich bei der ersten Deklaration schon psychisch und moralisch darauf eingestellt hat, die Funktion notfalls einlinken zu dürfen, und erst später mitgeteilt bekommt, daß er eben das doch nicht darf. Immerhin könnte ja da, wo jetzt „irgendwas" steht, bereits ein Funktionsaufruf einschließlich Symbolimport stattgefunden haben, und C und C++ sind so konzipiert, daß der Compiler das Programm in einem Pass übersetzen kann, ohne noch einmal zurückzugehen.

2.2.1.2.3 Typsicheres Linken und Link-Spezifikationen

Es war bereits mehrfach davon die Rede, daß „Symbole" an den Linker übergeben werden. Das sieht dann so aus, daß der Name des Objekts bzw. der Funktion dem Linker bekannt gemacht wird, versehen mit einem Vermerk, ob die Objekt-Datei das Symbol definiert (zusammen mit dem Symbolwert, also der Speicheradresse) oder benutzt (und einer Tabelle mit den Stellen, an denen dieses Symbol referiert wird). Allerdings wird der Name nicht ganz so exportiert, wie er definiert wird. Bei Variablen wird dem Bezeichner lediglich ein Unterstrich „_" vorangestellt, d. h. eine Variable „Daten" heißt für den Linker „_Daten". Dies ist eine allgemeine Konvention bei C-Compilern.

Es gibt übrigens im ANSI C-Standard eine mittlerweile ziemlich antiquierte Definition, nach der ein Linker nur die ersten sechs Zeichen eines Namens beachten und dabei auch nicht zwischen Groß- und Kleinbuchstaben unterscheiden muß. Demnach kann es theoretisch Systeme geben, auf denen die Variablen „String" und „stringlaenge" für den

Linker identisch sind. Mir ist aber kein solches System bekannt, und MaxonC++ beachtet wie jedes heute gebräuchliche C-System den ganzen Namen unter Unterscheidung von Groß- und Kleinbuchstaben.

In C++ ist es bekanntlich möglich, Funktionen zu überladen. Deshalb reicht es nicht, wenn man dem Linker lediglich den Namen einer Funktion übergibt, denn er muß ja in der Lage sein, die beiden Funktionen

```
void fun(int);
```

und

```
char *fun(char*);
```

zu unterscheiden. Dieses Problem wird gelöst, indem die Parameterliste in das Linkersymbol hineinkodiert wird:

Auf den Funktionsbezeichner (diesmal ohne vorangestellten „_“) folgen ein Unterstrich und eine Kodierung der Parametertypen. Dabei sind jedem Grundtyp ein oder zwei Buchstaben zugeordnet:

Typ	Code
char	c
signed char	Sc
unsigned char	Uc
short int	s
unsigned short	Us
int	i
unsigned int	Ui
long int	j
unsigned long	Uj
long long	l
unsigned long long	Ul
float	f
double	d
long double	D
void	v

Der Ergebnistyp der Funktion wird nicht kodiert, denn C++ erlaubt es nicht, zwei Funktionen mit identischen Parameterlisten, aber unterschiedlichem Rückgabetyp zu deklarieren. Ein Zeigertyp wird kodiert, indem dem Code des referierten Typs ein „P“ vorangestellt wird. Aus „char*“ wird also „Pc“ und „PPv“ ist ein Zeiger auf einen Zeiger auf „void“, also „void**“. Entsprechend stehen „R“ für Referenzen und „a“ (mit

zusätzlich kodierter Elementanzahl) für Vektoren. Die folgende Übersicht enthält auch schon einmal Codes für Datentypen, die in diesem Handbuch erst später eingeführt werden:

Pointer	P
Referenz	R
Vektor	A + Elementanzahl
const	C
volatile	V
Funktion	F + Parameterliste + f

Einige Datentypen können nur über ihren Namen kodiert werden:

Class, Struct, Union	Namenslänge (2 Ziffern) + Name
Enumeration	E + Namenslänge (2 Ziffern) + Name

Die Funktion „strcmp(char*, char*)" heißt dann z. B. für den Linker „strcmp_PcPc". Enthält der Funktionsname selbst einen Unterstrich, etwa „Input_data", wird dieser im Linkersymbol verdoppelt: „Input__data". Dadurch kann es nicht zu Kollisionen zwischen Unterstrichen im Funktionsnamen und dem Unterstrich zwischen Namen und Parameterliste kommen. Eine Ellipse „..." am Ende der Parameterliste wird durch ein angehängtes „e" dargestellt.

Das Kodieren der Parameterliste ist aber nicht nur eine Notlösung, um überladene Funktionen mit dem Linker behandeln zu können, sondern hat auch den angenehmen Effekt, daß auch über Modulgrenzen hinweg sichergestellt ist, daß Funktionsargumente den richtigen Typ haben. Deshalb nennt man diesen Mechanismus auch „typsicheres Linken".

ANSI C kennt kein Überladen, deshalb werden hier Funktionsnamen in der Regel genau wie Variablennamen kodiert. Auch MaxonC++ hält sich im C-Modus an diese Konvention. Dadurch wird es natürlich schwierig, C- und C++-Module zu verbinden, weshalb in C++ das Konzept der Link-Spezifikationen eingeführt wurde.

Bei einer Linkspezifikation folgt dem Schlüsselwort „extern" die gewünschte Programmiersprache als String, z. B.

```
extern "C" void test(int);
```

Diese Deklaration weist den Linker an, die Funktion „test" beim Linken nach C-Regeln zu behandeln, also „_test" statt „test_i". Jeder C++-Compiler kennt die Linkmodi „C" und „C++", in MaxonC++ gibt es auch noch „Asm" und „Pascal", wo dann das Symbol mit dem Bezeichner identisch ist und weder ein „_" am Anfang noch eine kodierte

Parameterliste bekommt. Da es aber langweilig ist, vor zehntausend Deklarationen jedesmal diese Spezifikation zu schreiben, kann man Deklarationen auch klammern:

```
extern "Asm"
{ void test(int);       // Externes Symbol: "test",
  int Anzahl, Maximum; // "Anzahl" und "Maximum",
  char* input();        // "input"
}
```

Wem es Spaß macht, der darf Linkspezifikationen auch ineinander verschachteln. Die Linkspezifikation müssen jeweils die erste Deklaration einer Funktion sein, sonst weiß der Compiler wieder nicht, was er soll. Bei weiteren Deklarationen bzw. Definitionen der Funktion muß man dann den Linkmodus nicht noch einmal angeben.

2.2.1.2.4 Die Schnittstelle zum Amiga-Betriebssystem

MaxonC++ kennt außer „C", „C++", „Asm" und „Pascal" noch den Linkmodus „Amiga", der eigentlich keiner ist. Dazu muß man erst einmal wissen, wie das Amiga-Betriebssystem aufgebaut ist.

Das Amiga OS besteht aus einer Sammlung von Libraries, also Bibliotheken oder Funktionssammlungen. Dabei vollzieht sich der Einsprung in eine solche Funktion etwas eigenartig. Zunächst einmal hat jede Library eine Basis-Struktur, die gewisse globale Daten der Bibliothek enthält. Wenn man eine Bibliotheksfunktion aufruft, wird verlangt, daß im Prozessorregister „a6" ein Zeiger auf eben diese Struktur steht. Vor der Basis-Struktur steht im Speicher eine Sprungtabelle, in der Sprunganweisungen zu den jeweiligen Bibliotheksfunktionen stehen.

Um eine Systemfunktion aufzurufen, lädt man einen Zeiger auf die Basis-Struktur nach „a6" und springt dann an einen gewissen (negativen) Offset unterhalb der Struktur. Die Argumente sind dabei in der Regel nicht auf dem Stack, sondern in Prozessorregistern zu übergeben, aber dazu mehr im folgenden Abschnitt. Jedenfalls werden Betriebssystemfunktionen in MaxonC++ nicht vom Linker eingebunden, sondern direkt angesprungen. Dazu benötigt der Compiler allerdings zwei Angaben: Die Basisadresse und den Offset der Funktion. MaxonC++ bietet dazu (auch im C-Modus) eine besondere Deklarationsweise, die an die Link-Spezifikationen angelehnt ist. Hinter dem Wortsymbol „extern" und dem Linkmodus „Amiga" folgt dabei zunächst der Name einer

Variablen, die die Basisadresse der Library enthalten soll. Hinter der eigentlichen Funktionsdeklaration ist dann der jeweilige Offset mit einem „=" anzugeben.
Bei der „Exec"-Library sieht das zum Beispiel so aus:

```
extern „AmigaLib" SysBase
{ void InitCode (register unsigned d0, register unsigned d1) =-0x48;
  void InitStruct (register APTR a1,register APTR a2, register unsigned
                   d0) =-0x4e;
  struct Library *MakeLibrary ( register APTR a0, register APTR a1,
                                register unsigned (*a2)(),
                                register unsigned d0,
                                register unsigned d1) =-0x54;
  void MakeFunctions ( register APTR a0, register APTR a1,
                       register unsigned a2) =-0x5a;
  struct Resident *FindResident (register UBYTE *a1) =-0x60;
  void InitResident (  register struct Resident *a1, register
                       unsigned d1) =-0x66;
  void Alert (register unsigned d7) =-0x6c;
  void Debug (register unsigned d0) =-0x72;
  void Disable (void) =-0x78;
  void Enable (void) =-0x7e;
  void Forbid (void) =-0x84;
  void Permit (void) =-0x8a;  // usw.
}
```

Wahrscheinlich werden Sie solche Deklarationen aber niemals selbst schreiben müssen, denn sie sind ja für alle Amiga-Libraries bereits vorhanden. Alternativ akzeptiert MaxonC++ auch die „#pragma amicall"-Schreibweise, wie sie von anderen C-Compilern bekannt ist. Diese wird im „Preprozessor"-Kapitel dargestellt.

Möglicherweise haben Sie sich oben über die seltsame Parametersyntax mit dem „register" gewundert. Das bietet dann auch schon die ideale Überleitung zum nächsten Abschnitt:

2.2.1.3 Daten in Registern

Jeder Prozessor verwaltet nicht nur das RAM, sondern hat darüber hinaus auch einige interne Prozessorregister, in dem in der Regel Zwischenergebnisse von Berechnungen o. Ä. abgelegt werden. Der Zugriff auf diese Register geht ganz erheblich schneller als eine RAM-Operation, und da die Prozessoren der 68000er-Familie mit Registern vergleichsweise üppig ausgestattet sind (ganz im Gegensatz zu diesen INTEL-Dingern), liegt der Wunsch nahe, häufig benutzte Variablen in den nicht anderweitig benutzten Registern abzulegen und so eine ordentlichen Geschwindigkeitssteigerung zu erzielen.

Das klingt zwar alles nach maschinennaher Bitpopelei, aber erfreulicherweise hat man im ANSI C-Standard eine flexible und maschinenunabhängige Schreibweise dafür eingeführt. Den Schlüssel dazu bietet eine Speicherklasse namens „register", die analog zu „extern", „auto" oder „static" benutzt wird.

Register-Variablen dürfen nur innerhalb von Funktionen deklariert werden. Ansonsten ist man in ihrer Verwendung weitgehend frei: Man darf beliebig viele Variablen beliebiger Typen als „register" deklarieren, wobei der Compiler dann entscheidet, was er wirklich in einem (und falls ja, in welchem) Register ablegt. MaxonC++ kann ganzzahlige und Pointer- bzw. Referenzvariablen in Registern unterbringen, wobei die numerischen Typen in Daten-, Zeigertypen in Adressregistern abgespeichert werden.

Auch hier gibt es wieder einen kleinen Unterschied zwischen C und C++: In ANSI C darf man nie die Adresse einer Variablen, die als „register" deklariert wurde, nehmen, also den unären Adressoperator „&" nicht anwenden. C++ ist da großzügiger: Wird die Adresse einer Registervariablen genommen, wird diese eben doch nicht in einem Prozessorregister, sondern ganz normal auf dem Stack abgelegt.

Registerdeklarationen bringen manchmal in der Tat einiges, die folgende Funktion läuft durch das „register" z. B. glatt doppelt so schnell:

```
void test()
{ register int i, j=0;

  for (i=0; i<500000; ++i)
    ++j;
}
```

Man sollte sich aber hüten, allzu ausgiebig von Registern Gebrauch zu machen, denn für praktisch alle Operationen braucht der Prozessor mindestens ein Register, und wenn zu viele davon für Variablen verwendet werden, muß er diese manchmal kurzfristig auf den Stack auslagern und nachher wieder laden, so daß man natürlich keinen Geschwindigkeitszuwachs erwarten kann. In jeder Funktion sollte man maximal vier ganzzahlige und zwei Pointervariablen als „register" deklarieren. Im Zweifelsfall sollte man in MaxonC++ möglichst ganz auf Register-Variablen verzichten und die Optimierungen einschalten, denn dann entscheidet der Compiler selbstständig, welche Variablen er in Register lädt.

Als kleine Erweiterung des C++-Standards bietet MaxonC++ die Möglichkeit, auch Parameter in Registern zu übergeben. Auch dies kann bisweilen zu gewissen Geschwindigkeitssteigerungen führen. Bei der Funktionsdeklaration ist dazu einfach das Symbol „register" vor die Parameter zu setzen, z. B.

```
int Add (register int i, register int j);
```

Dies entspricht aber nicht dem C++-Standard! Man sollte dieses Feature also möglichst vorsichtig einsetzen, da die Programme sonst nicht mehr portabel sind. Am besten benutzt man solche Parameter nur, wenn man fertige Funktionen, die ihre Argumente in Registern erwarten, irgendwie in ein Programm einbinden will, z. B. die Amiga-Betriebssystemfunktionen oder Assembler-Routinen, die sich ja ohnehin schlecht portieren lassen.

Es gibt sogar zwei Möglichkeiten, dem Compiler vorzuschreiben, in welche Register er die Parameter legen soll: Entweder bennent man die Parameter-Variable nach einem 68000er-Register, z.B.

```
void f(register int d0, register char *a2) ...
```

oder man setzt den gewünschten Registername mit „_" hinter das Schlüsselwort „register:"

```
void f(register _d0 int i, register _a2 char s) ...
```

Wenn man aber auf diese Weise dem Compiler schon Vorschriften macht, welche Register er für Variablen verwenden soll, sollte man sich vorher genau überlegen, was man da tut.

Für selbstgeschriebene C++ Funktionen ist dieses Feature so gut wie tabu, z. B. würde die oben deklarierte Funktion unter Umständen sehr langsam, weil sie möglicherweise das Register „d0" ständig braucht und den Inhalt der Register-Variablen dann jedesmal retten und anschließend wiederherstellen muß. Man sollte solche Register-Festlegungen ausschließlich zur Deklaration importierter Assembler- und Systemfunktionen benutzen.

2.2.1.4 Daten im CHIP-RAM

Die Architektur des Amiga ist etwas unkonventionell: Es gibt zwei, manchmal sogar drei verscheidene Klassen von RAM. Einerseits gibt es das Chip-RAM, auf das nicht nur der Prozessor, sondern auch die diversen DMA-Kanäle zugreifen können. Deshalb müssen hier alle Grafik- und Sounddaten stehen. Der übrige Speicher wird Fast-RAM genannt, weil er ausschließlich dem Prozessor zur Verfügung steht, weshalb Zugriffe auf diesen Speicher theoretisch schneller sein könnten. Auf normalen Amigas ist das aber nicht der Fall, so daß die Bezeichnung „Fast-RAM" eigentlich gelogen ist. Auf Amigas mit 32-Bit-Prozessoren (68020, 68030, 68040, ...) gibt es aber meist außerdem noch „echtes" Fast-RAM, nämlich unabhängig vom normalen Systembus getakteten, 32 Bit breiten Speicher. Deshalb muß man genaugenommen zwischem „echtem" und „falschem" Fast-RAM unterscheiden.

In der Praxis interessiert den Programmierer aber nur die Frage „Chip oder nicht Chip", denn er muß dafür sorgen, daß Grafikdaten im Chip-Memory abgelegt werden. Unter MaxonC++ ist das einfach: Es gibt dafür die Pragmas „chip" und „fast".

Alle nach der Zeile

```
#pragma chip
```

deklarierten statischen Daten werden ins Chip-Ram gepackt,

```
#pragma fast
```

schaltet wieder in den normalen Modus, in dem Daten irgendwo abgelegt werden, bevorzugt aber im Fast-Memory. Ein typsiches Beispiel ist eine Image-Struktur unter Intuition: Die Bitmap des Images dabei immer im Chip-Memory liegen.

```
#pragma chip

unsigned int BMap[] = // Bilddaten ins CHIP-RAM
{ 0x3FFC,
  0x7FFE,
  0xF00F,
  0xF00F,
  0xF00F,
  0xF00F,
  0x7FFE,
  0x3FFC
};

#pragma fast

// Die Image-Struktur selbst darf überall liegen:
struct Image MyImage = { 0, 0, 16, 8, 1, BMap, 1, 1, 0};
```

Auf automatische Daten hat „pragma chip" natürlich keinen Einfluß, denn diese werden generell auf dem Stack eingerichtet, und der liegt im allgemeinen nicht im Chip-Memory.

2.2.2 Typdefinitionen

Sicher haben Sie schon bemerkt, daß die Typschreibweisen in C einem ganz schön auf die Nerven gehen können, man denke nur an Dinger wie

```
int (*)[10] // Zeiger auf Vektor
```

Viel schöner wäre es doch, wenn man zusammengesetzte Datentypen mit einem Namen versehen und dann wie die Standardtypen verwenden könnte. Und genau das ist auch möglich, und zwar mit Typdefinitionen.

Eine Typdefinition wird mit dem Wortsymbol „typedef" eingeleitet. Rein syntaktisch sieht sie wie eine Variablendeklaration aus, wobei „typedef" als Speicherklasse, analog zu „static" oder „extern", benutzt wird, nur daß damit eben keine Variable, sondern ein Datentyp deklariert Ein Beispiel:

```
typedef int Vektor[10];
```

Nun steht der Bezeichner „Vektor" für den Datentyp „int[10]" und kann fortan in Deklarationen und Typbeschreibungen aller Art benutzt werden, z. B.

```
Vektor vec, *tra, matrix[5];
typedef Vektor *Vektorzeiger;
void AddVec (Vektor v1, Vektor v2, Vektor ergebnis);
```

Die Bezeichnung „Typdefinition" ist dabei etwas irreführend, denn in Wirklichkeit definiert man keinen neuen Typ, sondern führt lediglich einen neuen Namen für einen Datentypen ein. Das ist keine Spitzfindigkeit, sondern führt z. B. dazu, daß die Deklarationen

```
void Ausgabe(Vektor v);
void Ausgabe(int v[10]);
```

ein und dieselbe Funktion bezeichnen, denn die jeweiligen Parameter haben denselben Datentyp, auch wenn sie unter unterschiedlichen Typbezeichnungen deklariert wurden.

Einem Bezeichner kann mehrfach derselbe Typ zugewiesen werden, beispielsweise

```
typedef char *Strvec[42];
typedef char *Strptr;
typedef char Zeichen;
typedef Zeichen *Strptr;    // Nochmal definiert, OK
typedef Strptr Strvec[42]; // auch OK
typedef Strptr Strvec[41]; // ERROR
```

Außer zur Abkürzung oder zur Steigerung der Übersichtlichkeit kann man Typdefinitionen auch zur besseren Portabilität von Programmen einsetzen. Unter MaxonC++ ist ein „int" bekanntlich 32 Bit breit, aber es gibt durchaus auch Systeme, bei denen das nicht so ist (z. B. sind im MessyDos-Sektor noch 16-Bit-ints üblich). Deshalb ist es empfehlenswert, in einem Programm am Anfang zuerst einmal alle Datentypen zu deklarieren, z. B.

```
typedef int i32;
typedef unsigned u32;
typedef short i16;
typedef unsigned short u16;
```

Wenn man im restlichen Programm dann nur noch seine selbstdefinierten Typbezeichnungen verwendet, braucht man bei der Übertragung auf andere Compiler nur noch diese Typdeklarationen anzupassen.

Dabei ist es naheliegenderweise nicht möglich, Symbole wie „short" oder „unsigned" auf selbstdefinerte Datentypen anzuwenden, also nach den obigen Definitionen folgendes zu schreiben:

```
unsigned i32 x; // ERROR
u32 y;          // So ist's richtig
```

Eine Speicherklasse ist eine Eigenschaft eines bestimmten Datenobjekts, aber niemals Bestandteil eines Datentyps:

```
typedef static int SI; // ERROR
```

Typqualifizierungen sind dagegen durchaus Teil eines Datentyps, womit ich wieder einmal elegant zum folgenden Abschnitt übergeleitet habe:

2.2.3 Konstante Daten auf der Flucht

2.2.3.1 Die „const"-Qualifizierung

Ein Feature, das zuerst in C++ eingeführt und dann von ANSI C abgekupfert wurde, ist die Typqualifizierung mit den Symbolen „const" und „volatile". Man kann praktisch jeden Datentyp mit derartigen Qualifizierungen versehen und dann Objekte dieser Typen deklarieren, z. B.

```
const int i = 4711;
```

Der Wert der Variable „i", die den Datentypen „const int" hat, kann (so gut wie) nicht verändert werden, bleibt also immer der, mit dem sie initialisiert wurde. Den Versuch einer Wertzuweisung an „i" würde der Compiler nicht durchgehen lassen. Dabei weist er sogar ziemlich clevere Versuche zurück, etwa den folgenden:

```
void main()
{ const int i = 4711;

  int *ip;
  ip = &i;       // ERROR
  *ip = 98;

  const int *ip2;
  ip2 = &i;
  *ip2 = 99;     // ERROR
}
```

Nimmt man die Adresse einer „const"-Variablen, z. B. mit „&i", ist das Ergebnis ein Zeiger auf „const int". Den Versuch, einen solchen Zeigerwert „ip2", also einem normalen Zeiger auf „int", zuzuweisen, läßt der Compiler nicht durchgehen, denn anschließend könnte man ja den Wert von „i" über diesen Zeiger verändern. Der zweite Versuch wird auch fehlschlagen. Zwar wird „ip2" als Zeiger auf „const int" deklariert, so daß die Adresse von „i" an „ip2" zugewiesen werden kann. Wenn dann aber bei „*ip2" der Inhaltsoperator darauf angewendet wird, kommt ein L-Wert des Typs „const int" dabei heraus, und Wertzuweisungen daran sind wieder einmal verboten.

Umgekehrt darf man einen Zeiger auf etwas Konstantes durchaus mit nicht-konstanten Daten initialisieren:

```
void f()
{ int i, *pi;
  const int ci, *pci;

  pci = &i;  // OK
  pi  = &ci;   // ERROR
}
```

Übrigens bezeichnet

```
const T*
```

tatsächlich einen Zeiger auf ein konstantes „T". Einen konstanten Zeiger auf „T" deklariert man als

```
T *const
```

Beides kann man natürlich auch kombinieren:

```
int i;

const int *const p = &i; // Konst. Zeiger auf konst. "int"
```

Welchen Sinn hat nun dieses Gewurschtel mit dem „const"? Hauptsächlich dient dies dazu, anhand der Programmstruktur deutlicher auszudrücken, welche Daten wo verändert werden können. Beispielsweise kann man Vektoren nicht als solche, sondern nur in Gestalt von Zeigern darauf an Funktionen übergeben. Dadurch kann jede Funktion, die einen Vektor als Parameter hat, das als Argument übergebene Datenobjekt verändern, und man kann dann einem Funktionsaufruf nicht direkt ansehen, ob es dabei einen Seiteneffekt auf das Argument gibt. Sehen wir uns dagegen einmal die folgende kleine Funktion an, die nichts anderes tun soll, als einen String auszugeben:

```
#include <stream.h>

void Ausgabe (const char* st)
{ cout « st;
}
```

Der Programmierer kann innerhalb der Funktion „Ausgabe" nicht den Inhalt der Zeichenkette, auf die „st" zeigt, verändern, denn dann haut ihm der Compiler auf die Pfoten. Dann ist auch bei jedem Aufruf von „Ausgabe" klar, daß der Argument-String dadurch nicht verändert wird.

Die strengen Schutzvorschriften für konstante Daten gelten natürlich nicht nur für Zeiger, sondern auch für Referenzen. Eine Referenz auf einen nicht-konstanten Datentyp kann niemals mit der Adresse eines konstanten Objekts initialisiert werden, z. B.

```
const int i = 007;

int &ir1 = i;        // ERROR
const int &ir2 = i;  // OK
```

Rein syntaktisch ist es auch möglich, so etwas wie eine „konstante Referenz" analog zu den „konstanten Zeigern" zu deklarieren:

```
int &const irc = i;
```

In der Praxis hat man aber sowieso keine Möglichkeit, eine Referenz auf ein anderes Objekt als dem, mit dem sie initialisiert wurde, verweisen zu lassen, und deshalb ist es egal, ob man eine Referenz nun konstant deklariert oder in Peking eine Ente abstürzt.

Ein konstanter Referenz-Parameter wie

```
void f(const T &par);
```

ist eine echte Alternative zu „gewöhnlichen" Parametern wie

```
void g(T par);
```

vor allem dann, wenn „T" ein umfangreicher Datentyp ist, z. B. eine Struktur (siehe 2.5). Bei ersterem muß innerhalb der Funktion „f" bei jedem Zugriff auf „par" nachgesehen werden, wohin diese Referenz nun verweist, bei letzterem muß das Argument beim Funktionsaufruf auf den Stack kopiert werden. Der Programmierer sollte in solchen Situationen abschätzen, was wohl weniger Laufzeit kostet - im Zweifelsfall ist der Referenz-Parameter überlegen.

Der Datentyp

```
const char[20]
```

bezeichnet genaugenommen einen Vektor aus konstanten Zeichen und nicht einen konstanten Vektor. In der Praxis ist das aber natürlich dasselbe.

Eine „konstante Zeichenkette" der Länge „n" hat aus traditionellen Gründen den Typ „char[n]" und nicht etwa „const char[n]". Das liegt daran, daß die allererste C-Version „const" noch nicht kannte und es deshalb prinzipiell erlaubt ist, den Inhalt einer konstanten Zeichenkette zu verändern (siehe auch 2.1.2.5.1). Heutzutage ist eine solche Definition natürlich nicht mehr besonders sinnvoll, aber man wollte eben zu älteren C-Implementierungen kompatibel bleiben.

2.2.3.2 Flüchtige Daten: „volatile"

Mancher Compiler ist ziemlich schlau, und das kann Probleme geben. Beispielsweise kann man auf dem Amiga ziemlich rüde die Tastatur abfragen, nämlich an der Speicheradresse 0xbfec01:

```
void waitforspace()
{ unsigned char *key = (char*) 0xbfec01;

  while (*key != 0x7f); // Warte, bis "Space"-Taste gedrückt
}
```

Die Funktion wartet in einer „while"-Schleife darauf, daß die Leertaste gedrückt wird, indem immer wieder an der entsprechenden Speicheradresse nachgesehen wird, ob dort der entsprechende Code steht. Nun könnte ein Compiler ins Grübeln geraten und sich denken: „In dieser 'while'-Schleife wird ständig der Inhalt von '*key' benutzt, aber weder 'key' noch '*key' oder eine andere Variable verändert. Also wäre es doch dumm, '*key' jedesmal wieder auszuwerten. Ich lade diesen Wert lieber am Schleifenanfang einmal in ein Prozessorregister und benutze dann immer den Registerinhalt."

Das Ergebnis wäre zwar prinzipiell eine relativ schlaue Optimierung, aber eben falsch: Der Compiler ahnt ganz einfach nicht, daß der Inhalt von „*key" sich auch ohne Zutun des Programms verändern kann. Für solche Fälle gibt es die Qualifizierung „volatile",

die völlig analog zu „const" eingesetzt werden kann. In der Funktion „waitforspace" ist die Variablendeklaration zu ersetzen durch:

```
volatile unsigned char *key = (char*) 0xbfec01;
```

Nun weiß der Compiler, daß „key" auf einen Wert zeigen soll, der sich ganz unabhängig vom Programm verändern kann („volatile" heißt „flüchtig"), und daß er beim Ausdruck „*key" jedesmal wirklich an der jeweiligen Speicheradresse nachsehen muß.

Ganz penible Programmierer würden in diesem Fall übrigens „const" und „volatile" kombinieren:

```
volatile const unsigned char *key = (char*) 0xbfec01;
```

Aus der Speicherstelle „0xbfec01" kann man nämlich nur lesen, während Schreibzugriffe ergebnislos bleiben. Deshalb ist es für die (leider viel zu wenigen) Ästheten unter den C-Programmierern eine Selbstverständlichkeit, dieses dann auch durch ein zusätzliches „const" auszudrücken.

Da wir gerade bei solchem Brimborium sind: Noch schöner wäre natürlich

```
volatile const unsigned char *const key = (char*) 0xbfec01;
```

denn wir haben ja nicht das Bedürfnis, den Wert von „key", also die Speicheradresse, noch einmal zu verändern. Nun könnte der Compiler wieder etwas optimieren: Er weiß jetzt, daß „key" immer auf die Adresse „0xbfec01" zeigen wird, und braucht fortan den Wert von „key" nicht mehr zu betrachten, sondern kann von dem konstanten Wert ausgehen.

Eine typische Anwendung für die „volatile"-Qualifizierung ist auch die Kommunikation zwischen Tasks. Auf dem Amiga kann man bekanntlich, wenn auch nicht ganz so einfach wie unter UNIX, mehrere Prozesse gleichzeitig laufen lassen. Die Kommunikation zwischen parallelen Prozessen könnte man der Einfachheit halber über gemeinsame Variablen laufen lassen, und genau diese Variablen muß man dann unbedingt als „volatile" deklarieren, denn sonst weiß der Compiler nicht, daß ein anderer Prozess sie hinterrücks verändern kann.

Im großen und ganzen ist „volatile" aber ein „Low level"-Konzept, daß ganz einfach den Compiler davon abhalten soll, da zu optimieren, wo es unerwünscht ist, wogegen die Qualifizierung „const" ein sehr mächtiges Sprachkonstrukt ist, das fernab jeder Maschinennähe oder Compilerabhängigkeit dabei hilft, Datenflüsse im Programmtext auszudrükken. Ein disziplinierter, auf stilistische Sauberkeit bedachter Programmierer wird „const" überall da benutzen, wo es sinnvoll ist, aber nur selten in eine Situation geraten, wo er „volatile" benutzen kann oder sogar muß.

2.2.4 Inline-Funktionen

Jeder Funktionsaufruf stellt einen ziemlichen Aufwand dar. Zuerst muß der aufrufende Code auf dem Stack Platz für die Argumente einrichten und diese dort ablegen. Dann folgt ein „JSR"-Sprung in die Funktion, die zunächst den nötigen Speicherplatz für ihre eigenen Daten auf dem Stack reserviert und die Inhalte aller Prozessorregister, die sie verändert, irgendwo abspeichert. Am Ende der Funktion werden dann wieder die alten Registerinhalte hergestellt, die lokalen Variablen werden vom Stack entfernt und mit „RTS" wird in das aufrufende Programm zurückgesprungen, wo dann zu guter Letzt auch noch die Argumente wieder vom Stack genommen werden müssen. Da gerät man als Programmierer bisweilen in Gewissenskonflikte: Soll man nun eine pisselige kleine Funktion definieren oder den Inhalt dieser Funktion jedesmal in das Programm einsetzen? Ein typisches Beispiel ist die Maximum-Funktion:

```
int max(int a, int b)
{ if (a>b)
    return a;
  else
    return b;
}
```

Statt eines Funktionsaufrufs „max(x,y)" könnte man bekanntlich alternativ jedes mal „(x>y: x : y)" einsetzen, was zwar schneller, aber bestimmt nicht lesbarer wäre. Für solche Fälle hat C++ ein nettes kleines Feature: „Inline"-Spezifikationen.

Wenn Sie vor die erste Deklaration (und auf Wunsch auch vor die folgenden) einer Funktion das Schlüsselwort „inline" setzen, wird bei jedem Aufruf dieser Funktion die Funktionsdefinition gewissermaßen in den aufrufenden Quelltext eingesetzt. Syntaktisch oder semantisch wird das Programm dadurch nicht verändert: Man deklariert einfach

```
inline int max(int a, int b)
{ if (a>b)
    return a;
  else
    return b;
}
```

und kann „max" dann wie eine ganz gewöhnliche Funktion benutzen. Der Unterschied liegt lediglich im dafür generierten Code. Mit „inline" wird der Inhalt von „max" jedesmal in den aufrufenden Code hineinkopiert. Dadurch wird so einiges gespart: Es muß weder mit „JSR" in die Funktion hinein- noch mit „RTS" aus ihr herausgesprungen, es müssen in der Regel keine Registerinhalte gesichert und nachher wieder hergestellt werden (es werden in der Kopie von „max" nach Möglichkeit nur solche Register benutzt, die im aufrufenden Code gerade frei sind), und die Funktion muß für ihre Daten

keinen eigenen „Frame" auf dem Stack einrichten. Das Funktionsergebnis wird nicht notwendigerweise im Register d0, sondern dort, wo es gerade anfällt, zurückgegeben, und auch sonst hat der Compiler noch viele Optimierungsmöglichkeiten.

Eine Inline-Funktion wird also nicht nur einmal erzeugt, sondern in vielen Kopien, für jeden Aufruf eine. Es dürfte klar sein, daß Inlining Programme enorm aufblähen kann und man es deshalb damit nicht übertreiben sollte. Der Compiler darf aber prinzipiell das „inline" einfach ignorieren. MaxonC++ handhabt dies natürlich besser: Was „inline" sein soll, wird auch „geinlinet". Trotzdem kann es Situationen geben, in denen eine Inline-Funktion zusätzlich auf ganz gewöhnlich Art und Weise erzeugt werden muß, z. B. wenn man die Adresse einer solchen Funktion nimmt (siehe 2.3), wenn sie schon benutzt wird, bevor sie deklariert wird (schließlich kann der Compiler nicht hellsehen), oder wenn eine Inlinefunktion schlauerweise rekursiv ist (man kann eine Funktion ja nicht beliebig oft rekursiv in sich selbst hineinkopieren). All dies ist nämlich erlaubt, denn man kann eine Inline-Funktion ganz genau wie jede andere Funktion benutzen.

Wird eine Inline-Funktion aus einem der oben genannten Gründe (oder weil irgend so ein Compiler keine echten Inlines 'draufhat) real erzeugt, hat sie sinnvollerweise interne Linkage (wie mit „static" deklariert), so daß andere Module nichts davon merken, wenn eine Funktion in einer Übersetzungseinheit tatsächlich generiert wurde. Deshalb gelten für „inline" ähnliche Vorschriften wie für die Speicherklasse „static": Die Deklarationsreihenfolge

```
inline void f();
void f();
```

ist erlaubt,

```
void f();
inline void f(); // ERROR
```

hingegen nicht. Nebenbei bemerkt zählt „inline" nicht als Speicherklasse (wie „static" oder „extern"), sondern als „Funktions-Spezifierung" (wie auch „virtual", aber das kommt später). Diese Unterscheidung ist aber eher etwas für Standardisierungskommisionsmitglieder und Compilerbauer.

Abschließend ist als Zusammenfassung zusammenfassend zusammenzufassen, daß man mit „inline" zwar Laufzeit sparen kann, dies aber mit einer (oft erheblichen) Vergrößerung des Programms erkaufen muß. Deshalb sollte man „inline" nur bei wirklich winzigen Funktionen einsetzen.

2.3 Zeiger auf Funktionen

2.3.1 Wohin zeigen Zeiger auf Funktionen?

Im Abschnitt 2.2.4 wurde bereits angedeutet, daß jede Funktion, einmal abgesehen von „inline"-Deklarationen, einen Code hat, der erzeugt wird und in den eingesprungen wird. Natürlich liegt dieser Code dann irgendwo im Speicher und hat folglich auch eine Adresse. Genaugenommen nimmt eine Funktion natürlich einen ganzen Speicherbereich ein, hat also eine Anfangs- und eine Endadresse. Für ihre Benutzung ist aber lediglich die Einsprungadresse interessant, im allgemeinen identisch mit der Startadresse. Eben diese charakteristische Adresse einer Funktion kann man „nehmen" und benutzen, um die Funktion auf dem Umweg über diese Adresse aufzurufen.

Wie gewohnt, müssen wir auch hier zunächst wissen, wie man einen Zeiger auf eine Funktion, bzw. eine solche Zeigervariable, deklariert. Sei „f" eine Funktion, die ein „int" als Parameter hat und „double" zurückgibt. Dann wissen Sie natürlich längst, wie man ein solches „f" deklariert:

```
double f(int);
```

Nun möge „fp" eine Variable sein, die auf eine Funktion eben dieser Art zeigt. Die deklariert man dann so:

```
double (*fp)(int);
```

Sie erinnern sich vielleicht: Die Klammern um „*fp" sind nötig, weil „fp" sonst eine Funktion ist, die einen Zeiger auf „double" zurückgibt, also im Prinzip das gleiche Theater wie bei dem „Zeiger auf Vektor" und „Vektor von Zeigern".

Weiter geht es dann ganz wie gewohnt: Man weist „fp" die Adresse einer passenden Funktion, also z. B. „f", zu. Mit „passend" ist hier gemeint, daß Parameterliste und Ergebnistyp der Funktion mit dem von „fp" referierten Funktionstyp übereinstimmen müssen. Die Adresse einer Funktion nimmt unser bekannter Adressoperator „&":

```
fp = &f; // Nun zeigt "fp" auf Funktion "f"
```

Das Gegenteil vom Adress- ist der Inhaltsoperator, folglich kommen wir mit „*fp" an die Funktion, auf die „fp" zeigt, und können sie ganz normal mit einer geeigneten Argumentliste aufrufen, z. B. so:

```
double d1 = (*fp)(42);
```

Auch hier sind die Klammern um „*fp“ nötig, denn sonst würde wieder einmal die Argumentliste stärker binden als der Inhaltsoperator: „*fp(42)“ ruft eine Funktion „fp“ auf (die es natürlich nicht gibt) und wendet auf deren Ergebnis den Operator „*“ an.

2.3.2 Noch ein paar Anmerkungen zu Syntax und anderen Details

Die Regeln, wie man mit Deklarationen wie

```
char*(*xxx)(char*);
```

klarkommt, wurden ja bereits in Abschnitt 2.1.2.3 genannt, und wenn Sie die beherzigen, werden Sie schnell herausfinden, daß hier „xxx“ als ein Zeiger auf eine Funktion, die sowohl als Parameter und als Ergebnis einen Zeiger auf „char“ hat, deklariert wird. Auch wissen Sie (hoffentlich) noch, daß man Datentypen wie Variablendeklarationen schreibt, nur daß man den Variablennamen wegläßt. Das soll hier, anhand eines Beispiels aus der Praxis demonstriert werden:

```
void konfus()
{ void *p;
  int a,b,c;

  (**(void(**)(...))p)(a,b,c); // Nanu?
}
```

Was ist das??? Also, wenn man die mysteriöse Zeile einmal betrachtet, stellt man fest, daß man auf oberster Ebene zwei aufeinanderfolgende Klammerausdrücke hat, deren rechter, „(a,b,c)“, nach allen C-Sprachregeln nur eine Argumentliste sein kann. Folglich ist der rätselhafte linke Teil der Anweisung ein Ausdruck, der einen Zeiger auf eine Funktion darstellt. Schaun' mer doch ma', was da so in der Klammer steht: Da sind zwei Inhaltsoperatoren „*“, ein weiteres Klammerpaar mit seltsamem Inhalt und schließlich der Bezeichner „p“. Dieser innere Klammerausdruck, „(void(**)(...))“, kann nur ein Cast sein, also eine explizite Typumwandlung, folglich muß „void(**)(...)“ ein Datentyp sein. Wenn wir darin einfach einen Bezeichner einfügen,

```
void (**x)(...);
```

erhalten wir die Deklaration eines Zeigers auf einen Zeiger auf eine Funktion, die beliebige Parameter und Ergebnistyp „void“ hat. Also wird in unserem Beispiel der Bezeichner „p“ in genau jenen Typ verwandelt, anschließend wird zweimal der Inhaltsoperator angewandt, so daß wir die Funktion erhalten, auf die der Zeiger verweist, auf den „p“ zeigt. Zum Abschluß wird diese Funktion mit den Argumenten „(a,b,c)“ aufgerufen. Wenn man es so sieht, ist das doch ganz einfach...

Es gibt allerdings eine gewisse Konvention, nach der der Name einer Funktion identisch ist mit dem Zeiger auf Startadresse dieser Funktion, ähnlich wie der Name eines Vektors der Adresse des ersten Vektorelements entspricht. Deshalb kann man es sich in der Regel sparen, den Operator „&“ auf Funktionen anzuwenden, z. B.

```
void f();

void main()
{ void (*fp)();

  fp = main;   // Statt "&main"
  fp = f;      // Statt "&f"

  (*fp)();     // Funktionsaufruf
}
```

In ANSI C ist sogar ausschließlich diese Schreibweise erlaubt, während C++ es dem Programmierer überläßt, ob er hier das „&“ setzen will oder nicht.

Wenn aber der Name einer Funktion mit einem Zeiger auf diese Funktion identisch ist, besteht ein Funktionsaufruf immer aus zweierlei: einem Zeiger auf eine Funktion und einer Argumentliste. Deshalb ist es in Folge des obigen Features zusätzlich auch erlaubt, beim Funktionsaufruf über Pointer den Operator „*“ wegzulassen:

```
void f();

void main()
{ void (*fp)() = f;

  fp();  // Statt "(*fp)()"
}
```

Dadurch sieht ein Funktionsaufruf über Zeiger syntaktisch wie ein gewöhnlicher Funktionsaufruf aus. Diesmal erlauben sowohl ANSI C als auch C++ beide Schreibweisen. Somit hängt es auch hier wieder von Ihrem persönlichen Geschmack ab, ob Sie ein „*“ setzen wollen oder nicht.

2.3.3 Zeiger auf Überladene Funktionen

C++ kennt bekanntlich das schöne Feature des Überladens von Funktionen. Die daraus resultierenden Probleme beim Funktionsaufruf werden durch einen umfangreichen Satz von Matching-Regeln gelöst, so daß der Compiler anhand der Typen der Argumente stets die richtige Funktion herausfinden kann. Wenn man aber die Adresse einer Funktion

nimmt, gibt man ja nur ihren Namen, aber keine Argumente an. Wie kann man also sauber angeben, welche Funktion nun bei Überladung gemeint ist? Dafür hat man in C++ eine verblüffend einfache Lösung gefunden.

Betrachten wir als Beispiel einmal die beiden Funktionen

```
int f(int);
double f(double);
```

Der Ausdruck „&f" (oder einfach „f", ganz nach Geschmack) stellt nun nicht mehr einen Zeiger auf eine bestimmte Funktion dar, sondern hat den ganz eigenen Datentyp „Zeiger auf die überladene Funktion f" und keinen bestimmten Wert. Dieser Ausdruck kann aber in einen „passenden" Funktionszeigertyp umgewandelt werden, nämlich (klar) in die beiden Typen „int (*)(int)" und „double (*)(double)". Einen Zeiger auf eine überladene Funktion kann man aber (im Gegensatz zu allen anderen Zeigertypen) nicht z. B. in „void*" umwandeln, denn dann kann der Compiler ja nicht aus dem Zusammenhang schließen, welche Funktion „f" gemeint ist.

Also enthält das folgende Programm zwei richtige und zwei falsche Initialisierungen:

```
int f(int);
double f(double);

void main()
{ int (*fp1)(int) = f;   // OK, da eindeutig
  void (*fp2)(int) = f;  // ERROR, da kein passendes "f"

  void *vp1 = main;      // OK, "main" nicht überladen
  void *vp2 = f;         // ERROR, "f" ist überladen
}
```

Die von „vp1" referierte Funktion könnte man übrigens nicht direkt aufrufen, sondern muß den Zeigerausdruck erst casten, aber wie das geht, stand ja schon in 2.3.2.

2.3.4 Eine Anwendung: „qsort" und „bsearch"

Weniger erfahrene Programmierer werden jetzt vielleicht fragen, was das mit den Zeigern auf Funktionen eigentlich soll. Um es einmal ganz vornehm auszudrücken, sind Funktionszeiger eine elegante und schnelle Möglichkeit, nicht nur Daten, sondern auch Verfahren und Methoden abzuspeichern. Da der typische C-Programmierer aber eher ein praktisch denkender, bodenständiger Mensch ist, soll hier ein kleines Beispiel angeführt werden.

Es gibt Tage im Leben eines Programmierers, da muß man ganz allgemein Daten irgendwie sortieren. Die Algorithmenforschung hat im Laufe der Zeit etliche Sortierverfahren hervorgebracht, deren Darstellung ich mir hier aber ersparen möchte. Eines davon ist jedenfalls als „Quicksort" bekannt, und der Nomen, äh, Name ist ein Omen, denn dieses Verfahren ist wirklich ziemlich schnell. Als C-Programmierer müssen Sie es noch nicht einmal selbst implementieren, denn Quicksort steht als Standardfunktion zur Verfügung!

Dabei muß der Funktion irgendwie in Form eines Arguments gesagt werden, um welche Art von Daten es sich handelt und nach welchen Kriterien diese sortiert werden sollen. Genaugenommen braucht die Funktion folgende Angaben:

1. Wo liegt der zu sortierende Vektor (Anfangsadresse)?
2. Wie viele Elemente hat er?
3. Wie groß ist jedes einzelne Element? (Der genaue Datentyp ist egal, nur der Speicherplatzbedarf interessiert.)
4. Unter welchen Umständen soll ein Element im sortierten Vektor vor einem anderen stehen?

Die drei ersten Argumente kann man mit einem Pointer und zwei Zahlen erledigen, das letzte Argument kann sinnvoll nur in Form eines Zeigers auf eine Funktion realisiert werden.

Die Quicksort-Funktion ist in „<stdlib.h>" folgendermaßen deklariert:

```
void qsort(void *base, unsigned n, unsigned size, int
(*cmp)(const void*, const void*) );
```

Dabei bedeuten:

„base": Zeiger auf Vektoranfang
„n": Elementanzahl
„size": Elementfröße
„cmp": Zeiger auf eine Funktion, die zwei Vektorelemente vergleicht und einen negativen Wert liefert, wenn das erste Argument im Sinne der Sortierreihenfolge „kleiner" als das zweite ist, bzw. Null, wenn beide Elemente gleich sind, oder einen positiven Wert, wenn das erste Element „größer" als das zweite ist.

Als Beispiel wollen wir ganz einfach einen Vektor von hundert „int"-Werten sortieren und dann ausgeben. Um uns die Eingabe der Daten zu ersparen, ziehen wir einfach mit der Funktion „rand()", ebenfalls aus „<stdlib.h>", 100 Zufallszahlen.

Vor allem brauchen wir aber eine Vergleichsfunktion, die Zeiger auf zwei „int"-Objekte erhält und diese dann vergleicht. Formal muß man die Argumente leider als „void"-Pointer deklarieren:

```
int CompareInt(const void *p1, const void *p2);
```

Deshalb muß jede Benutzung der Parameter „p1" und „p2" mit einem Clast „(int*)" versehen werden. Zufälligerweise liefert eine Subtraktion von „int"-Werten gerade das gewünschte Ergebnis: negativ bei „kleiner", Null bei „gleich" und positiv bei „größer". Also sieht unser kleines Programm so aus:

```
#include <stdlib.h>  // Für "qsort" und "rand"
#include <stream.h>

int CompareInt(const void *p1, const void *p2)
{ return *(int*)p1 - *(int*)p2;
}

void main()
{ int Vek[100], i;

  for (i=0; i<100; ++i)   // Vektor initialisieren:
    Vek[i] = rand();      // Irgendwelche Zufallszahlen

  qsort(Vek, 100, sizeof(int), CompareInt);  // Sortieren

  // Daten ausgeben (10 Zahlen pro Zeile):
  for (i=0; i<100; ++i)
    cout « Vek[i] « (i%10 == 9 ? "\n" : " ");
}
```

Wozu ist ein sortierter Vektor gut? Einerseits ist es natürlich oft ganz einfach schöner, wenn man Daten sortiert ausgibt. Vor allem kann man Daten in einem sortierten Vektor wesentlich besser suchen. Stellen Sie sich z. B. einmal das Telefonbuch vor. Da die Einträge dort alphabetisch sortiert sind, muß man nicht das Ganze Buch von vorn an durchlesen, um einen Namen zu finden. Man schaut zuerst irgendwo in das Buch und stellt fest, ob der Eintrag, den man gerade gefunden hat, vor oder hinter dem gesuchten steht, und sucht dann entweder vor oder hinter der aktuellen Position weiter. Dies wiederholt man dann so lange, bis man den gewünschten Eintrag gefunden hat.

Auch der Computer kann in sortierten Daten so suchen. Dabei wird der zu durchsuchende Bereich bei jedem Schritt halbiert, weshalb man dieses Verfahren „binäres Suchen" nennt. Auch dafür gibt es in „<stdlib.h>" eine Funktion, nämlich „bsearch":

```
void *bsearch(const void *key, const void *base, unsigned
n, unsigned size,
int (*cmp)(const void*, const void*));
```

„key" ist dabei ein Zeiger auf ein Datenobjekt, das im Vektor zu suchen ist, die übrigen Parameter sind wie bei „qsort". In unserem Beispiel könnte man nach dem Sortieren mit

```
int j = 42;
void *position = bsearch(&j, Vek, 100, sizeof(int),
                         CompareInt);
```

prüfen, ob die Zahl „42" zufällig im Vektor vorkommt. Wenn ja, zeigt das Funktionsergebnis auf das entsprechende Vektorelement, und man könnte z. B. mit

```
(int*)position - base
```

den Index berechnen. Kommt „42" nicht vor, liefert „bsearch" das Ergebnis Null.

2.4 Aufzählungen

2.4.1 Deklaration

Die bis hier erwähnten Standard-Datentypen, also diverse Arten von Zahlen, Zeichen und Zeichenketten, sind zwar ganz nützlich, aber oft nicht ausreichend, um Daten zu repräsentieren. Wenn man die Daten einer Person abspeichern will (Achtung, immer brav das Bundesdatenschutzgesetz beachten!), können wir den Namen (eine Zeichenkette) oder das Geburtsjahr (ganzzahlig) mit den Standardtypen behandeln, aber wie drücken wir den Familienstand oder das Geschlecht in Daten aus?

Die Standardmethode dafür ist, eine numerische Kodierung einführen, etwa „0 heißt ledig, 1 verheiratet, 2 geschieden..." und solche Daten dann als Zahlen abzuspeichern. Schön ist das aber nicht gerade, denn diese nackten Zahlen sagen irgendwie nichts aus. Deshalb gibt es in C das Konzept der „Aufzählungstypen".

Eine Aufzählung oder „Enumeration" wird nicht aus den Standardtypen abgeleitet, sondern stellt einen ganz neuen, eigenen Datentyp dar. Deshalb kann man einen Aufzählungstypen nicht einfach beschreiben wie z. B. einen Pointertypen, sondern muß ihn deklarieren, und zwar zweckmäßigerweise unter einem bestimmten Namen.

Die Deklaration eines Aufzählungstyps besteht aus dem Schlüsselwort „enum“ und dem (optionalen) Typnamen, z. B.

```
enum Farbe;
```

Mit so einer Deklaration kann man aber noch nichts anfangen, vor allem keine Objekte dieses Typs deklarieren. Jeder „enum“-Typ braucht sinnvollerweise eine Deklaration, bei der festgelegt wird, welche Werte er umfassen soll. Ein Beispiel:

```
enum Farbe { Rot, Gruen, Blau };
```

Der Aufzählungstyp „enum Farbe“ umfaßt jetzt die drei Werte „Rot“, „Grün“ und „Blau“. Jeder dieser Werte ist eine numerische Konstante, die durch einen Bezeichner repräsentiert wird. Das folgende Programmfragment demonstriert die Anwendung von Aufzählungen:

```
#include <stream.h>

enum Familienstand { ledig, verheiratet, geschieden, verwitwet
};

void Heirat (enum Familienstand P1, enum Familienstand P2)
{ if (P1 == verheiratet || P2 == verheiratet)
    cout « "Ein Partner ist schon verheiratet!\n";
  else
    cout « "Geht in Ordnung.\n";
}

void main()
{ enum Familienstand f1, f2;

  f1 = ledig;

  // ...

  Heirat (f1, f2);

  // ...
```

```
  switch (f1)
  { case ledig:
      cout « "ledig";
        break;
      case verheiratet:
        cout « "verheiratet";          break;
      case geschieden:
      case verwitwet:
        cout « "nicht mehr verheiratet";
        break;
    }

  // ...
}
```

Man beachte vor allem, daß Aufzählungswerte Bezeichner und keine Strings sind. Folglich stehen sie im Programm ohne Hochkommata und können nicht direkt über „cout" ausgegeben werden.

Man kann auch Typ- und Variablendeklaraton vermischen:

```
enum Antwort {Ja, Nein} A1 = Ja, A2 = Nein, A3;
```

Diese Zeile deklariert den Aufzählungstyp „Antwort", die beiden Konstanten „Ja" und „Nein" und die drei Variablen „A1", „A2" und „A3". Auf diese Weise ist es auch möglich (allerdings meist nicht sehr sinnvoll), namenlose Aufzählungstypen zu deklarieren:

```
enum {Ja, Nein} A1 = Ja, A2 = Nein, A3;
```

Der hier definierte namenlose Datentyp mit den beiden Werten „Ja" und „Nein" kann nicht noch einmal für Deklarationen benutzt werden: Wenn man eine Zeile wie

```
enum {Ja, Nein} A4;    // ERROR: "Ja"+"Nein" redefined
```

hinzufügt, beschwert sich der Compiler, daß die beiden Bezeichner „Ja" und „Nein" überdefiniert werden. Jede Aufzählung darf höchstens einmal definiert werden, auch nicht zweimal mit exakt denselben Werten.

2.4.2 Typnamen und Typlabel

Etwas seltsam ist, daß bei Aufzählungstypen in C das Schlüsselwort „enum" immer mitgeschleift werden muß: Nach der Deklaration

```
enum X {a, b, c,};
```

gibt es einen Datentypen namens „enum X", während „X" allein kein Typname, sondern lediglich ein sog. „Typlabel" ist. Originellerweise kollidieren die Typlabel eines Scopes nicht mit anderen Bezeichnern, so daß folgende Deklaration durchaus möglich ist:

```
enum X {a, b, c};  // Typdefinition

enum X X;          // Variablendeklaration
```

Die erste Zeile deklariert einen Aufzählungstyp mit dem Typlabel „X", während in der zweiten Zeile eine Variable namens „X" deklariert wird, die obendrein auch noch den Typ „enum X" hat... ganz schön verrückt!

Man kann dieses Feature aber auch benutzen, um das ewige „enum" zu vermeiden:

```
typedef enum X {a, b, c} X;
```

Auf diese Weise deklariert man sowohl ein Typlabel „X" als auch einen gleichnamigen Typbezeichner, und fortan kann man wahlweise mit

```
enum X x1;
```

als auch mit

```
X x2;
```

Variablen deklarieren - jedenfalls in ANSI C. In C++ haben Sie es wieder einmal einfacher: Nachdem man einen Aufzählungstyp deklariert hat, kann man das „enum" fortan weglassen. Ein Typlabel kann in C++ wie ein Typname verwendet werden. Es gilt aber trotzdem noch, daß gleichlautende Typlabel und Typnamen nicht kollidieren.

2.4.3 Aufzählungen und Zahlen

Naturgemäß kann ein Computer mit Zahlen viel besser umgehen als mit Bezeichnern oder Zeichenfolgen, und so verwundert es nicht, daß Aufzählungstypen intern als Zahlen dargestellt werden. In MaxonC++ ist ein Aufzählungstyp stets 32 Bit breit.

Ein originelles Feature ist, daß Aufzählungsdaten ohne explizite Konvertierung als „int"-Werte benutzt werden können:

```
enum Farbe { rot, gruen, blau };

int i = gruen;
```

Der Compiler zählt von Null an, also ist im obigen Beispiel „rot" 0, „gruen" 1 und „blau" gleich 2, und die Variable „i" wird mit eins initialisiert. In der umgekehrten Richtung muß wieder einmal zwischen C und C++ unterschieden werden: In ANSI C darf ein ganzzahliger Ausdruck an eine Aufzählungs-Variable zugewiesen werden (wenn auch der Compiler eine Warnung ausgeben darf), in C++ geht das leider nicht, z. B.

```
int i = gruen;      // OK
enum Farbe f1 = i; // In C erlaubt, in C++ ERROR
```

Diese Einschränkung in C++ liegt daran, daß im allgemeinen nur wenige Zahlen gültige Werte eines Aufzählungstyps sind. Im Beispiel umfaßt der Typ „Farbe" nur drei verschiedene Werte, dargestellt durch die Zahlen 0 bis 2. In C soll der Programmierer selbst darauf achten, daß er keine andere Zahlenwerte an eine „Farbe"-Variable zuweist, während C++ wieder einmal etwas pingeliger ist und derartige Versuche generell verbietet.

Entsprechend sieht es mit der Zuweisungskompatiblität zwischen verschiedenen Aufzählungstypen aus:

```
enum X {x1, x2} x;
enum Y {y1, y2} y = x;    // Nur in C, nicht in C++
```

In allen diesen Beispielen ist in C++ die Typwandlung mit einem Cast zu erzwingen:

```
enum Y {y1, y2} y = (Y) x; // So geht's auch in C++
```

Manchmal will man, aus welchem Grund auch immer, die Zahlenwerte, die der Compiler den Aufzählungskonstante zuordnet, selbst bestimmen. Auch das ist möglich, indem man in der Typdefinition die gewünschten Werte angibt:

```
enum Farbe {rot = 1, gruen, blau = 4};
```

Wird einem Namen kein Wert zugeordnet, so wie hier „gruen", zählt der Compiler einfach vom letzten vorher angegebenen Wert weiter, so daß „gruen" hier mit dem Wert „2" belegt wird. Die Konstante kann von jedem nach „int" wandelbaren Typ sein, also ein Ganzzahl- oder Aufzählungstyp. Es muß sich natürlich um einen konstanten, also schon zur Compilezeit berechenbaren Wert handeln.

Solche „enum"-Definitionen sind die „sauberste" Methode, in C ganzzahlige Konstanten zu deklarieren. Ein Beispiel:

```
enum { N = 100 };

int Vec[N];
```

```
void main()
{ for (int i=0; i<N; i++)
  Vec[i] = 0;

// usw.
}
```

Wenn Sie im obigen Programm die Anzahl der Vektorelemente ändern wollen, müssen Sie nur noch die „enum"-Deklaration entsprechend verändern. Zwar ist „N" formal keine ganze Zahl, sondern hat einen eigenen, namenlosen Aufzählungstyp, aber das macht in der Praxis nichts, denn Aufzählungstypen wie der von „N" können überall da eingesetzt werden, wo man auch „int" benutzten kann.

Dementsprechend kann man mit Aufzählungsdaten auch genau wie mit „int"-Zahlen rechnen:

```
enum E {a1, a2, a3, a4};

void main()
{ enum E e1 = a1;

  int i = e1+a2; // OK

  e1 += 1;       // ERROR
}
```

Der Ausdruck „e1 += 1" ist in C++ nicht erlaubt, denn er entspricht bekanntlich „e1 = e1 + 1". Dabei wird die Summe „e1 + 1" aber sinnvollerweise als „int" ausgewertet, und eine Zuweisung der Form „enum = int" ist in C++ bekanntlich verboten.

2.5 Strukturen

2.5.1 Grundlagen

Es ist eine scheinbar triviale, aber in Wirklichkeit geradezu revolutionäre Idee, Daten, die zusammengehören, auch formal in einem Datentyp zusammenzufassen. Zum Beispiel besteht eine komplexe Zahl aus einem Realteil und einem Imaginärteil, die beide jeweils durch eine Fließkommazahl dargestellt werden können. Nun wäre es unschön, in einem Programm für jede komplexe Zahl gleich zwei Variablen zu deklarieren. Statt dessen definiert man sich einen Verbund-Datentyp namens „Komplex" (oder so ähnlich), der aus zwei Fließkommazahlen besteht. Das könnte man notfalls auch noch mit Vektoren realisieren:

```
typedef double Komplex[2];  // Element [0] für Realteil,
                            //    [1] Imaginärteil
Komplex x1 = {1,0}, x2 = {2,2};
```

Aber hier haben wir ja auch das Glück, daß Real- und Imaginärteil mit demselben Datentyp, nämlich „double", dargestellt werden können. Wenn wir aber Personendaten abspeichern wollen, und zwar Vorname („char[20]"), Nachname („char[30]") und Geburtsjahr („int"), geht das schon nicht mehr, denn naturgemäß haben alle Elemente eines Vektors den gleichen Typ. Genau hier setzt das Konzept der Datenstrukturen ein, das übrigens auch die Grundlage für die objektorientierte Programmierung darstellt.

Eine Struktur ist ein Datentyp, der Einzeldaten beliebiger Typen zusammenfaßt. Die Elemente einer Struktur werden nicht wie bei Vektoren mit einer Nummer, sondern mit einem Namen bezeichnet. Eine Struktur für die oben erwähnten Personendaten könnte man so deklarieren:

```
struct Person
{ char Vorname[20], Nachname[30];
  int Geburtsjahr;
};
```

Jetzt ist dem Compiler der Datentyp „struct Person" bekannt, und wir können uns Variablen deklarieren:

```
struct Person P1, P2;
struct Person *P_Zeiger = &P1;
```

und so weiter. Jedes Objekt des Typs „struct Person" besteht aus drei Datenelementen, die man in C++ auch als „Member" bezeichnet. Member werden formal genau wie Variablen deklariert. Der Zugriff auf einen Member erfolgt, indem man an einen

geeigneten Ausdruck, z. B. einen Variablennamen, einen Punkt „.“ und den Namen des Members anhängt:

```
void main()
{ struct Person P;

  cout « "Bitte geben Sie ein:\n";

  cout « "Vorname:   "; cin » P.Vorname;
  cout « "Nachname:  "; cin » P.Nachname;
  cout « "Jahr:      "; cin » P.Geburtsjahr;
  cout « "\n";

  if (P.Geburtsjahr > 1992 || P.Geburtsjahr < 1880)
    cout « "Das Jahr ist offensichtlich nicht richtig!\n";
  else
  { int Alter = 1992 - P.Geburtsjahr;

    cout «  "Hallo " « P.Vorname « " " « P.Nachname « ", Du bist " « Alter «
            " Jahre alt.\n";
  }
}
```

Der Punkt „.“ zählt, genau wie ein Index oder eine Argumentliste, zu den Postfix-Operatoren und damit zur höchsten Bindungsklasse. Deshalb ist

```
*x.y
```

äquivalent zu

```
*(x.y)
```

und nicht etwa zu

```
(*x).y
```

Letzteres tritt übrigens so oft auf, daß es dafür eine Abkürzung gibt:

```
x->y
```

Um das ein wenig zu illustrieren, folgt auch prompt ein Beispiel:

```
struct S
{ char c1, *c2; };  // S hat zwei Member: char c1 und char *c2
```

```
void main()
{ struct S s1,s2, *sp;

  // ...

  sp = &s1;           // sp zeigt auf s1

  (*sp).c1 = *(s2.c2); // Ist äquivalent zu:
  sp->c1 = *s2.c2;
}
```

„sp->c1" ist der Member „c1" in der Struktur, auf die der Zeiger „sp" verweist, während „*s2.c2" das Zeichen ist, auf das der Member „c2" der Struktur „s2" zeigt.

2.5.2 Definitionen und Deklarationen

Die Deklaration einer Struktur beginnt, wie Sie sicher schon bemerkt haben, immer mit dem Wortsymbol „struct". Anschließend ist alles optional, nur die Reihenfolge ist wichtig:

1. Strukturname, auch „Typlabel" genannt
2. Typdefinition, also die Deklaration der Member
3. Variablendeklarationen

Am Ende muß immer ein Semikolon stehen. Gehen wir nun halbwegs systematisch vor:

```
struct;
```

Diese Zeile ist ausnahmsweise nicht erlaubt, denn mindestens einer der drei oben angeführten Punkte muß folgen.

```
struct Name;
```

Diese Deklaration teilt dem Compiler mit, daß es einen Strukturtyp namens „Name" gibt. Die eigentliche Typdefinition kann später noch folgen, muß aber nicht. Nach einer solchen Mini-Deklaration kann man mit dem Typ „struct Name" nur solche Dinge tun, bei denen der Compiler nichts näheres über „struct Name" wissen muß. Man kann Zeiger auf diesen Typ deklarieren, z. B. „struct Name *p;", aber keine Objekte dieses Typs selbst („struct Name s;"). Man darf Funktionen deklarieren, die „struct Name" zurückgeben, aber keine Funktionen mit Parametern dieses Typs. Als kleines Extra darf man Objekte dieses Typs als „extern" deklarieren und somit importieren.

```
struct Name { int Memb1; char Memb2; };
```

Dies ist wohl die gebräuchlichste Form der Strukturdefinition. Jede Struktur darf beliebig oft deklariert werden (also so etwas wie „struct Name;"), aber höchstens einmal definiert werden. Man darf sie auch nicht zweimal absolut identisch definieren. Nach der Definition weiß der Compiler alles über den Strukturtyp, nämlich die Namen und Typen seiner Member und vor allem den Speicherplatzbedarf der Struktur. Die Deklaration

```
struct { int Memb1; char Memb2; };
```

ist zwar rein syntaktisch erlaubt, aber wenig sinnvoll. Hier wird ein Datentyp definiert, aber man kann nichts damit anfangen, denn er hat keinen Namen. Bei Strukturtypen zählt übrigens Namensgleichheit, d. h. zwei Strukturtypen mit absolut identischen Membern sind für den Compiler völlig unterschiedliche Datentypen.

```
struct Name objekt1, vektor[20], *zeiger;
```

Hier werden drei Variablen auf gewohnte Weise deklariert. Der Typ „struct Name" kann allgemein wie ein Standardtyp oder ein „typedef"-Bezeichner benutzt werden. Für die Deklarationen von „objekt1" und „vektor[20]" muß der Strukturtyp bereits definiert sein, die Pointervariable „*zeiger" kann auch schon vor der endgültigen Definition von „struct Name" erfolgen.

Man kann Strukturdefinition und Variablendeklaration auch kombinieren:

```
struct Name { int Memb1; char Memb2; } objekt1;
```

Diese Zeile deklariert gleichzeitig den Strukturtyp und eine Variable dieses Typs. Dabei kann man aber auch den Strukturnamen weglassen:

```
struct { int Memb1; char Memb2; } objekt1;
```

Auch hier werden ein Strukturtyp und eine Variable deklariert, aber der Strukturtyp hat keinen Namen und kann deshalb nie wieder für weitere Deklarationen benutzt werden.

Wenn Ihnen das andauernde „struct" auf die Nerven geht, können Sie eine Struktur auch so deklarieren:

```
typedef struct Name { int i; } TypName;
```

Fortan können Sie den Typ wahlweise „struct Name" oder einfach „TypName" nennen. Das kann man auch so weit treiben, daß die Struktur selbst keinen Typlabel mehr hat:

```
typedef struct { int i; } TypName;
```

Offenbar gibt es viele Ästheten oder Schreibfaule, die von Deklarationen wie der obigen reichlich Gebrauch machen. Deshalb sah man bei der Einführung von C++ Handlungsbedarf und legte fest, daß ein Typlabel in C++ wie ein „typedef"-Name benutzt werden kann, sofern es dabei nicht zu Kollisionen mit anderen Bezeichnern kommt:

```
struct Name { int i; };

Name objekt1;          // Nur in C++ erlaubt
struct Name objekt2;   // In C nötig, in C++ erlaubt
```

Die Deklaration der Member innerhalb einer Strukturdefinition ist formal identisch mit einer Variablendeklaration, außer daß man hier natürlich keine Speicherklassen verwenden kann (okay okay, in C++ können Member als „static" deklariert werden, aber das gehört hier noch nicht hin). Also können Member natürlich auch selbst wieder Strukturen sein, wie im folgenden Beispiel:

```
struct Datum
{ short int tag, monat, jahr; };

struct Name
{ char vorname[20], nachname[30]; };

struct Person
{ struct Name name;
  struct Datum geburt;
};

struct Person P1, P2, P3; // "struct" kann hier in C++
                          // auch weggelassen werden

#include <string.h>  // Immer Ärger mit dem Stringshandling von C

void main()
{ P1.geburt.monat = 6;
  strcpy(P1.name.vorname,"Helmut");

  P2 = P1;            // kopiert ganze Struktur
  P3.name = P1.name; // kopiert nur eine "Unterstruktur"

  // usw.
}
```

An diesem Beispiel sehen Sie auch, daß eine „struct" in C ein „echter" Datentyp ist, den man mit einer simplen „="-Zuweisung kopieren kann, während das bei Vektoren bekanntlich etwas krampfig ist (sieht man hier an „strcpy"). Man kann auch Strukturen direkt an Funktionen übergeben und nicht etwa bloß Zeiger darauf wie bei den Vektoren.

2.5.3 Initialisierung

Eine Struktur kann mit einer Struktur gleichen Typs initialisiert werden - einleuchtend, denn man kann solche Strukturen ja auch normal einander zuweisen. Daneben gibt es wie bei Vektoren die Möglichkeit einer Listeninitialisierung. Das ist nur bei Strukturen erlaubt, die keine Basisklassen, Konstruktoren oder private Member haben, aber da das alles Features sind, die in diesem Handbuch erst später behandelt werden, braucht Sie das vorerst nicht zu kümmern. Jedenfalls sind die Initialisierungswerte für die einzelnen Member in der richtigen Reihenfolge anzugeben und mit geschweiften Klammern „{ }" einzuschließen:

```
struct Anschrift
{ char Name[40], Anschrift[30];
  unsigned short PLZ;
  char Ort[25];
};

Anschrift A1 = {"Helmut Kohl",
                "Bundeskanzleramt",
                5300, "Bonn" };
```

Wenn man zu wenige Daten angibt, werden die übrigen Member mit Nullen initialisiert; bei zu vielen Daten beschwert sich der Compiler.

Bei verschachtelten Strukturen haben Sie wieder, wie bei mehrdimensionalen Vektoren, die Wahl, ob Sie Listen ineinander verschachteln oder nicht:

```
struct Datum
{ short int tag, monat, jahr; };

struct Name
{ char vorname[20], nachname[30]; };

struct Person
{ Name name;
  Datum geburt;
};

Person   P1 = { { "Karl", "Baumann" }, {26, 7, 1931} },
         P2 = { "Hein", "vom Deich", 24, 5, 66 };
```

Dasselbe gilt bei Vektoren von Strukturen:

```
struct S
{ char Name[20];
  int Jahr;
};

S Liste1[ ] = { {"Djikstra", 1940 }, {"Seville", 1976},
                {"Davidson", 1903 } };

S Liste2[ ] = { "Dingenskirchen", 1952, "Werwohl", 1968,
                "Nocheiner", 1967,"Schmuddel", 1972 };
```

2.5.4 Gültigkeitsbereiche und Scope-Regeln

Für die Typlabel von Strukturen gilt im Prinzip das gleiche wie bei „enum"-Typen: Sie haben einen eigenen Gültigkeitsbereich innerhalb jedes anderen Scopes. Dadurch sind Deklarationen wie

```
struct X X;
```

möglich: „X" ist der Name eines Objekts und „struct X" ein Typname, so daß Verwechslungen prinzipiell ausgeschlossen sind.

Aber jetzt kommt der Hammer: Da Strukturen und Klassen (was fast dasselbe ist) das zentrale Konzept der objektorientierten Features von C++ sind, hat man hier in C++ einige gravierende Änderungen gegenüber ANSI C vorgenommen. Zunächst einmal darf man, wie bereits unter 2.5.2 angedeutet, das Schlüsselwort „struct" (und auch „class" und „union", dazu später mehr) weglassen und das Typlabel direkt als Typnamen verwenden, natürlich nur dann, wenn keine Kollisionen auftreten:

```
struct X { int x; };

X V;            // OK
struct X W;     // Das auch
X X;            // Sogar das geht!
X Y;            // Error
struct X Z;     // So ist's jetzt richtig
```

Ein vernünftiger Programmierer wird normalerweise Kollisionen von Typ- und Variablennamen vermeiden. Die Entwickler des Amiga-Betriebssystems gehörten offensichtlich nicht dazu, denn in den offiziellen Amiga-System-Includefiles findet man etliche derartige Kollisionen, z. B. muß man bei der „struct DateStamp" aus „libraries/

dos.h" das „struct" immer dazusetzen, weil in „clib/dos_protos.h" eine Funktion gleichen Namens deklariert wird... wirklich, ausgesprochen sauberer Stil!

Wirkliche Probleme gibt es mit der Kompatibilität zwischen C und C++ aber bei einem anderen Feature, den struktureigenen Gültigkeitsbereichen. Zu jeder Struktur gehört ein eigener Scope für die Namen der Member. Das bedeutet, daß die folgenden Deklarationen

```
struct S1 { int x; };
struct S2 { char x; };
double x;
```

nicht kollidieren: das erste „x" liegt im Gültigkeitsbereich von „S1", das zweite im Scope von „S2" und das dritte auf Dateiebene. Es ist an jeder Stelle des Programms klar, welcher von den drei Bezeichnern mit „x" gemeint ist. Übrigens war das vor ANSI C nicht so direkt klar, denn man hatte ganz einfach vergessen, dies in den allerersten C-Spezifikationen festzulegen. Deshalb könnte es durchaus noch uralte C-Compiler geben, die obiges Programm nicht akzeptieren. Aber das kann Ihnen ja egal sein, denn Sie haben schließlich viel Geld für einen C++- und ANSI-C-Compiler hingeblättert und sind deshalb an derartigen Anachronismen nicht mehr interessiert.

Ach ja, ich sollte vielleicht mal langsam zur Sache kommen und verraten, wo denn nun hier der Unterschied zwischen C und C++ liegt. Also, weil Strukturen ja das Grundkonzept für die objektorientierte Programmierung darstellen, hat man ihnen in C++ einen ganzen Scope spendiert, während in ANSI C nur die Membernamen im Scope der Struktur liegen. Der Sinn dieser kryptischen Aussage wird wahrscheinlich am ehesten an einem Beispiel klar:

```
struct S1
{ int i;
  struct S2 *s2zeiger;   // VORSICHT!!!
};

struct S2
{ int j; };
```

Für den C-Compiler ist hier alles klar: Die Namen „i" und „s2zeiger" gehören in den Scope von „S1", aber das Typlabel „S2", das ja kein Membername, sondern eben ein Typlabel ist, gehört auf Dateiebene. Damit ist klar, daß der Member „s2zeiger" auf den Strukturtyp „S2" zeigt, der gleich anschließend definiert wird, und so ist das Ergebnis das erwartete.

In C++ sieht das anders aus: Hier kann man innerhalb einer Struktur nicht nur Member, sondern auch (aber das gehört schon zu den objektorientierten Features) Funktionen, Variablen oder eben auch Datentypen deklarieren, die dann lokal zur Struktur gehören und von außen normalerweise nicht sichtbar sind. Nun kommt also ein Compiler an und

stellt fest, daß innerhalb von „S1" ein bisher völlig unbekannter Datentyp „struct S2" benutzt wird. Da der Datentyp neu ist, handelt es sich offenbar um eine Deklaration, und das Verhängnis nimmt seinen Lauf. Der Compiler nimmt gemäß der C++-Sprachdefinition an, daß es sich um einen zu „S1" lokalen Datentypen handeln muß. Wenn dann irgendwann später auf Dateiebene der Typ „struct S2" definiert ist, denkt der Compiler nicht im Traum daran, daß dieses „S2" und das aus „S1" irgend etwas miteinander zu tun haben könnten. Die Folge: Der Member „s2zeiger" aus „S1" zeigt auf einen nicht definierten Typ, und

```
S1 s;
s.s2zeiger->j = 0;
```

führen zu einer Fehlermeldung, denn „j" ist ja Member des globalen „S2" und nicht etwa des lokalen.

Deshalb muß man in C++ stets folgendermaßen deklarieren:

```
struct S2;

struct S1
{ int i;
  struct S2 *s2zeiger; // Jetzt OK
};

struct S2
{ int j; };
```

Da „S2" jetzt zu dem Zeitpunkt, wo es in „S1" benutzt wird, bekannt ist, braucht der Compiler auch nicht mehr zu unterstellen, daß es sich um einen lokalen Datentyp handelt. Es gilt also wieder die Daumenregel, daß man in C++ alles erst möglichst sauber deklarieren muß, bevor man es benutzen kann.

Sie halten das hier diskutierte Thema für etwas esotherisch? Von wegen! Die Amiga-Standardincludes machen ausgiebig von solchen Pointereien zwischen Strukturen Gebrauch! Watt nu? Soll man bei jeder Benutzung dieser Includes vorher erst mit „#pragma -" in den C-Modus schalten, oder was?

Keine Panik: MaxonC++ gehört zu den relativ schlauen Compilern. Wird innerhalb einer Struktur ein anderer Strukturtyp deklariert und benutzt, aber nicht wirklich definiert, und außerhalb dieser Struktur ein Typ gleichen Namens definiert, so betrachtet MaxonC++ diese beiden Typen als identisch. Also schluckt MaxonC++ das Beispiel auch ohne vorherige Deklaration von „struct S2". Durch dieses Feature gerät MaxonC++ natürlich in Konflikt mit dem gültigen Standard, und es kann passieren, daß unter MaxonC++ entwickelte Programme sich mit anderen Compilern nicht übersetzen lassen, aber andererseits ist dieses Feature durch Stroustrup abgesegnet (im „Anachronismen"-

Kapitel des „Annotated C++ Reference Manual“ von 1989, Abschnitt 18.3.5), und so erschien es uns sinnvoll, es zu implementieren.

2.5.5 Unionen

Unionen sind ein kleines Feature, mit dem man manchmal etwas Speicherplatz sparen kann. In C++ sind sie eigentlich überholt, denn Polymorphie läßt sich viel eleganter auf objektorientiertem Wege durch Vererbung erzielen. Aber wie so manche Altlast von C wurden auch Unionen in C++ aufgenommen, um die heilige Kuh namens „Kompatiblität“ nicht ohne Not zu schlachten, und ungeachtet aller Vorbehalte sollen sie auch hier erwähnt werden.

Syntaktisch werden Unionen genau wie Strukturen behandelt, nur daß das Schlüsselwort „struct“ durch „union“ zu ersetzen ist. Der Trick ist dabei, daß alle Member einer union denselben Speicher belegen und man deshalb immer nur einen davon benutzen kann, z. B.

```
union U { int Nummer; char *Name; };

U u1, u2;

void main()
{ u1.Nummer = 42;
  u2.Name = "Frood";
}
```

In diesem Beispiel muß also irgendwie inhärent klar sein, daß bei „u1“ nur der Member „Nummer“ und bei „u2“ ausschließlich „Name“ benutzt werden. Die jeweils anderen Member haben undefinierte Werte, und beim Versuch, sie auch noch zu initialisieren, werden die ersten Definitionen wieder überschrieben, denn „Nummer“ und „Name“ einer Struktur liegen ja im selben Speicher.

Damit dürfte ansatzweise deutlich geworden sein, daß man mit Unionen Speicherplatz sparen kann, wenn man von mehreren Membern immer nur jeweils einen pro Datenobjekt benötigt. Nehmen wir einmal an, wir wollen Daten von verschiedenen Computern abspeichern. Bei allen Computern wollen wir die Bezeichnung und die Speicherkapazität abspeichern, aber bei Amigas sollte man angeben, wieviel vom RAM CHIP-Mem ist, bei MS-DOSen sollte der Prozessortyp nicht verschwiegen werden, und bei ATARI-Computern braucht man unbedingt einen String mit einem Kommentar, was wir davon halten. Also entwerfen wir folgende Datentypen:

```
enum Typ {Amiga, MessyDOS, Atari};
```

```
struct Rechner
{ char Name[20];
  int kBytes;
  enum Typ wasistes;
  union
  { int ChipMem;
  char Prozessor[6];
    char *Kommentar;
  } Spezial;
};
```

Man beachte, daß wir hier einen namenlosen „union"-Typen innerhalb der Struktur definieren. Man hätte die Unionsdefinition auch herausziehen, auf Dateiebene mit einem Typlabel sauber definieren und dann innerhalb der Struktur benutzen können, aber so geht's natürlich auch. Anhand des Members „wasistes" kann man jedenfalls jedem Datenobjekt ansehen, welcher Member der Union aktuell ist, z. B. so:

```
#include <stream.h>
#include <string.h>

void Ausgabe(Rechner &r)
{ cout « "Bezeichnung:  " « r.Name « "\n";
  cout « "Speicher:     " « r.kBytes « " kBytes\n";

  switch (r.wasistes)
  { case Amiga:
      cout « "(davon " « r.Spezial.ChipMem « " kByte CHIP)\n";
      break;
    case MessyDOS:
      cout « "Prozessor:    " « r.Spezial.Prozessor « "\n";
      break;
    case Atari:
      cout « "Unsere Meinung dazu: "«r.Spezial.Kommentar«"\n";
      break;
  }
  cout « "\n";
}
```

Auch Unionen können mit „Listen" initialisiert werden, allerdings kann die „Liste" hier nur genau einen Initialisierungswert enthalten, und zwar für den ersten Member der Union:

```
void main()
{ Rechner  R1 = { "Amiga 500", 3*1024, Amiga, { 512 } },
    R2 = { "ST", 1024, Atari },
    R3 = { "Taiwan Trashy 0815", 640, MessyDOS };
```

```
  // Bei „R2" und „R3" muß die „Spezial"-Information
  // nachträglich initialisiert werden:
  R2.Spezial.Kommentar = "Ach neee, muß ja nicht sein...";
  strcpy (R3.Spezial.Prozessor,"486sx"); // Hihihi...

  Ausgabe(R1);
  Ausgabe(R2);
  Ausgabe(R3);
}
```

Wo hier die Speicherplatzersparnis liegt? Eine Union braucht so viel Speicher wie ihr größter Member. Das „int" beim Amiga und „char*" beim Atari belegen jeweils 4 Bytes, während der String bei MessyDOS 6 Zeichen lang ist. Also ist die ganze Union 6 Bytes lang, was einer Ersparnis von 8 Bytes entspricht.

Zu guter letzt sei hier noch ein etwas abgefahrenes Feature von C++ erwähnt: namenlose Unionen. Wird eine Union ohne Typlabel einfach nur definiert, ohne daß eine Variablendeklaration o. ä. folgt, ist dies in Wirklichkeit eine Deklaration mehrerer Variablen, die alle an der gleichen Adresse liegen:

```
#include <stream.h>

static union
{ int i;
  char c;
  double d;
};

// Die drei globalen Variablen "i", "c" und "d" liegen
// alle an der gleichen Adresse! Der Beweis:

void main()
{ cout « int(&i) « "\n";
  cout « int(&c) « "\n";
  cout « int(&d) « "\n";
}
```

Das „static" ist hier aus rein formalen Gründen nötig, weil die Sprachdefinition das fordert. Eine solche Deklaration ist z. B. dann sinnvoll, wenn ein Programm mehrere große Vektoren hat, aber immer nur höchstens einen davon braucht. So etwas dürfte aber doch ziemlich selten vorkommen.

Namenlose Unions sind eher innerhalb von Strukturen nützlich:

```
enum Typ {Amiga, MessyDOS, Atari};

struct Rechner
{ char Name[20];
  int kBytes;
  enum Typ wasistes;
  union
  { int ChipMem;
    char Prozessor[6];
    char *Kommentar;
  };
};

Rechner R1;
```

Jetzt liegen die drei Member „ChipMem", „Prozessor" und „Kommentar" nicht mehr in einer Union verborgen, sondern direkt in der Struktur „Rechner". Also kann man sich auch das ewige „Spezial" sparen, z. B.

```
R1.Kommentar = "Ei der daus!".
```

Die Struktur „Rechner" sieht also noch genau so aus wie vorher, nur daß der Dummy-Member „Spezial" entfällt und dessen Member direkt in der Struktur stehen, und zwar auch wieder im selben Speicherbereich, der natürlich nicht mit dem von „Name", „kBytes" oder „wasistes" kollidiert. Bemerkenswerterweise sind auch die Scope-Regeln dem angepaßt, d. h. die Bezeichner der Member der namenlosen Union könnten mit gleichlautenden Bezeichnern in der überliegenden Struktur zusammenrappeln, und das ist so ja auch sinnvoll.

Der Vollständigkeit halber sei hier schon darauf verwiesen, daß die meisten objekt-orientierten Features bei Unionen nicht unterstützt werden: Eine Union kann keine Basisklassen haben und keine Member, die einen Konstruktor oder Destruktor besitzen.

2.5.6 Bitfelder

Auch Bitfelder sind so ein Low-Level-Sprachkonstrukt, mit dem man bisweilen ein paar Byte sparen kann. Bekanntlich bietet C dem Programmierer ganzzahlige Datentypen in verschiedenen, für den jeweiligen Rechner typischen Breiten. Kein heute existierernder, ernstzunehmender Rechner unterstützt jedoch Zahlen von weniger als acht Bit, also einem Byte. Das liegt daran, daß der Zugriff auf den Speicher immer nur byteweise erfolgen kann, während das Lesen oder sogar Schreiben von einzelnen Bits einen recht hohen Aufwand darstellt.

Oft stellen 8-Bit-Werte aber schon eine Verschwendung dar. Ein typisches Beispiel sind Flags, also einfache boolesche Werte, die ohnehin nur die Werte „0" und „1" annehmen können, wofür ein Bit vollkommen reicht. Manchmal will man auch mehrere Zahlen, deren Wertebereich stark begrenzt ist, auf engstem Raum abspeichern, etwa in einem Langwort (32 Bit) vier Zahlen mit 20, 10 und zweimal je einem Bit unterbringen - die Alternative wäre eine Struktur aus einem „int", einem „short int" und zwei „char", die genau doppelt so viel Speicher benötigt.

Formal ist ein Bitfeld eine Struktur, bei der hinter jedem Member die Anzahl der benötigten Bits anzugeben ist, z. B.

```
struct Beispiel1
{ int a: 20, b: 10;
  unsigned int c: 1, d: 1;
};
```

Die Member eines Bitfelds dürfen ausschließlich ganzzahlige Datentypen haben. Es ist nicht erlaubt, Bitfelder mit normalen Strukturen zu vermischen, also für einige Member Bitbreiten anzugeben und für andere nicht. Ein Bitfeld kann nicht mit einer Elementliste initialisiert werden. Überhaupt ist vieles bei Bitfeldern implementationsabhängig und weder vom ANSI-C-Standard noch in der gängigen C++-Literatur verbindlich definiert. Der Zugriff auf einen Bitfeld-Member geschieht syntaktisch wie bei einer gewöhnlichen Struktur - aber Vorsicht, der Rechner benötigt dazu einen erheblich höheren Aufwand als bei einem gewöhnlichen Strukturzugriff. Das macht die Programme dann sowohl länger als auch langsamer. Deshalb sollte man es sich immer sehr genau überlegen, ob man wirklich Bitfelder einsetzen sollte. Oft wird dadurch der Programmcode dermaßen aufgebläht, daß die Speicherplatzersparnis bei den Daten wieder „ausgeglichen" wird, einmal ganz zu schweigen von der erheblich langsameren Zugriffsgeschwindigkeit auf die Daten!

Es ist erlaubt, hier namen- und typlose Member zu definieren:

```
struct BF2
{ int a: 3;
  :2;
  int b: 4;
};
```

Dadurch werden mitten zwischen „a" und „b" zwei Bits freigehalten. So etwas braucht man manchmal, wenn man ein vorgegebenes Layout übernehmen muß. Apropos Layout: MaxonC++ belegt die Bits von links nach rechts. Im obigen Beispiel belegt die Struktur „BF2" zwei Bytes, wobei die Bits 5 bis 7 des ersten Bytes „a" enthalten, die Bits 3 und 4 desselben Bytes unbelegt bleiben und „b" in dem drei untersten Bits des ersten sowie dem obersten Bit des zweiten Bytes liegt. Ein namenloser Member der Breite Null belegt nicht etwa Null Bits, sondern bewirkt eine Ausrichtung auf die nächste Bytegrenze:

```
struct BF3
{ int a: 3;
  :0;    // Macht das erste Byte "voll"
  int b: 8;
};
```

Dadurch liegt hier „b“ genau in einem Byte, so daß ein Zugriff auf „b“ zufällig keinen zusätzlichen Overhead erfordert.

Es ist nicht möglich, direkt oder indirekt oder wie auch immer die Adresse eines Bitfeld-Members zu nehmen:

```
struct S1 { int flag1:1, flag2:2 };

S1 s1;

int *p = &s1.flag1;    // ERROR
int &r = s1.flag2;     // ERROR
```

Generell kann ich nur raten, möglichst die Pfoten von Bitfeldern zu lassen, meist lohnt es sich ganz einfach nicht.

2.6 Dynamisch organisierte Datenstrukturen

2.6.1 Die Idee

Bisher haben wir alle Variablen, die ein Programm braucht, schon beim Schreiben des Programms deklariert. In der Praxis ist das allerdings keineswegs ausreichend, denn oft ist vorher nicht abzusehen, wieviel Daten das Programm zur Laufzeit benötigen wird. Nehmen wir an, Sie schreiben einen Editor und haben dabei eine Datenstruktur namens „struct Zeile“, die jeweils eine Zeile des Texts repräsentiert. Nun wissen Sie aber natürlich nicht, wie lang der Text sein wird, den der Benutzer einmal eingeben wird. Und genau hier kann man oft Hobby- und echte Programmierer unterscheiden: der Hobbyprogrammierer legt sich einen Vektor von, sagen wir einmal, zehntausend „Zeile“-Strukturen an und geht davon aus, daß der Benutzer nicht mehr als eben jene zehntausend Textzeilen eingeben wird, falls doch, hat er eben Pech gehabt. Andererseits belegt der Editor dann natürlich immer konstant einige Megabytes, aber das fällt ja nicht auf, schließlich hat man ja sowieso nur ein Programm gleichzeitig laufen. Auf dem Amiga geht das natürlich nicht, denn der Amiga-User wird sich „freuen“, wenn ein Task den gesamten Speicher belegt.

Die einzig professionelle Methode, einen Editor zu programmieren, ist die benötigten Datenobjekte für Zeilen, Zeichen usw. bei Bedarf dynamisch zu erzeugen und wieder zu löschen, wenn sie nicht mehr benötigt werden, weil z. B. der Benutzer etwas aus dem Text gelöscht hat. Das beste Beispiel hierfür ist der in MaxonC++ verwendete Editor EDWARD. Und um solche Techniken soll es in diesem Abschnitt gehen, wobei wir auch einen eleganten Übergang zur objektorientierten Programmierung erhalten werden. Vorher sollten Sie sich allerdings vergewissern, daß Sie die Sache mit den Pointern auch wirklich verstanden haben, denn sonst wird es sehr, sehr schwer!

2.6.2 „new" und „delete"

Es soll also davon die Rede sein, wie man Datenobjekte zur Laufzeit des Programms dynamisch erzeugt und später auch wieder löscht. Sie waren es bisher gewohnt, daß ein Datenobjekt mit einer Variablen identisch ist und deshalb immer einen Namen hat. Ein dynamisch erzeugtes (man sagt neudeutsch auch „alloziertes") Objekt hat natürlich keinen Namen, denn dann müßte man es ja irgendwie im Programmtext deklarieren und hätte nichts gewonnen. Es liegt also nur irgendwo im Speicher 'rum und hat deshalb eine Adresse, und Adressen verarbeitet man in C mit Zeigern.

Rein praktisch erzeugt man in C++ ein neues Datenobjekt mit einem „new"-Ausdruck. Das Schlüselwort „new", gefolgt von einer Typbezeichnung, richtet an geeigneter Stelle im Speicher ein neues Datenobjekt dieses Typs ein und liefert als Ergebnis einen Zeiger darauf, z. B.

```
int *ip = new int;
```

Der Zeiger „ip" verweist jetzt also auf ein Datenobjekt „*ip" des Typs „int", das vorher noch nicht existierte. Aber Vorsicht, es kann durchaus sein, daß nicht genug Speicher zur Verfügung stand! In diesem Fall liefert „new" den Wert 0. Deshalb gehört hinter jeden „new"-Aufruf eigentlich eine „if"-Abfrage, die feststellt, ob auch wirklich Speicher reserviert wurde. Viele Programmierer sind leider zu faul dazu und vertrauen ganz einfach darauf, daß „new" klappt.

Aber auch für solche Leute gibt es in C++ein nettes Gimmick: Im Include-File „<new.h>" wird eine Funktion namens „set_new_handler" deklariert, die als Argument eine „void"-Funktion ohne Parameter erwartet. Diese Funktion wird fortan automatisch aufgerufen, wenn eine Speicherreservierung mit „new" fehlschlug, und man kann in dieser Funktion versuchen, die Situation zu retten, oder doch wenigstens das Programm mit „exit" abbrechen.

Zum Löschen eines mit „new" erzeugten Objekts dient die „delete"-Anweisung:

```
delete ip;
```

Nun ist der Wert von „ip“ laut Standard undefiniert und der Speicherbereich, in dem vorher „*ip“ lag, wieder als unbelegt und frei definiert. In MaxonC++ zeigt „ip“ nach dem „delete“ übrigens immer noch auf dieselbe Adresse, aber es kann durchaus Implementationen geben, bei denen „ip“ dabei auf „0“ oder einen anderen Wert gesetzt wird. Wenn man versucht, ein Objekt zu löschen, das nicht mit „new“ erzeugt wurde, steigt das Programm aus. MaxonC++ löscht am Programmende alle dynamischen Daten, die noch existieren.

Laut C++-Standard ist ein dynamisches Objekt nach seiner Erzeugung absolut undefiniert. MaxonC++ initialisiert den Speicher mit lauter Nullen, aber darauf kann man sich nicht generell verlassen.

Man kann Datenobjekte beliebiger Typen mit „new“ erzeugen - mit einer Ausnahme, nämlich Referenzen. „new int&“ liefert also einen Fehler, denn Referenzen sind in C++ strenggenommen keine Objekte, oder zumindest Objekte zweiter Klasse. Natürlich darf man auch keine Strukturen erzeugen, die noch nicht definiert sind, denn dann weiß der Compiler ja noch nicht, wieviel Speicherplatz dafür benötigt wird.

2.6.3 Dynamisch erzeugte Vektoren

Vielleicht fragen Sie sich jetzt schon, was der Quatsch soll. Eine Zeile wie „int *ip = new int“ ist in der Tat ziemlich nutzlos, denn es kommt doch wohl auf dasselbe heraus, ob man eine „int“-Variable oder einen Zeiger darauf deklariert. Interessant werden dynamisch generierte Objekte erst dann, wenn man für eine Pointervariable jede Menge dynamischer Objekte alloziert und diese Anzahl vor der Ausführung des Programms noch nicht bekannt ist. Am einfachsten geht das in C++ über Vektoren, denn natürlich lassen sich auch solche Objekte dynamisch erzeugen:

```
void f()
{ char *cp;

  cp = new char[1000];
}
```

Man beachte, daß in C ein Vektor mit einem Pointer auf sein erstes Element identisch ist. Deshalb deklarieren wir hier „cp“ als einfachen Zeiger auf „char“ und nicht als Zeiger auf einen „char“-Vektor.

Nun ist auch das nicht so furchtbar schlau, denn man hätte hier einen entsprechend großen Zeichenvektor direkt deklarieren und sich die Sache mit dem Zeiger sparen können (einmal davon abgesehen, daß das hier erzeugte Objekt existiert, bis es gelöscht wird, also auch über die Ausführung der Funktion „f“ hinaus). Praktischerweise akzeptiert C++ an dieser Stelle auch einen beliebigen variablen Ausdruck als Vektorgröße:

```
void g(int groesse)
{ char *cp = new char[groesse];
}
```

Na, ist das was? Das Programm kann jetzt erst einmal feststellen, wieviel Speicher es wohl brauchen wird, und dann einen entsprechend großen Vektor einrichten. Es gibt allerdings eine kleine Einschränkung: Bei mehrdimensionalen Vektoren darf ausschließlich der erste Indexbereich variabel sein:

```
void h(int i, int j)
{ new double[i][j];     // FALSCH
  new double[10][j];    // FALSCH
  new double[i][10];    // OK
}
```

Der Grund ist ganz einfach der, daß ein C-Programm beim Zugriff auf ein Vektorelement nicht wissen muß, wie viele Elemente der Vektor hat (der Programmierer soll gefälligst selbst darauf achten, daß er die Vektorgrenzen nicht überschreitet), wohl aber die Größe eines Vektorelements benötigt, denn alle Vektorelemente liegen im Prinzip hintereinander im Speicher, so daß man die Adresse des n-ten Elements aus der Vektoranfangsadresse erhält, indem man dazu das n-fache der Elementgröße addiert. Also stellt es für das Programm kein Problem dar, auf ein Element eines „beliebig großen Vektors von zehnelementigen double-Vektoren" zuzugreifen, denn ein „zehnelementiger double-Vektor" ist ganz einfach so groß wie zehn „double"-Zahlen. Umgekehrt kann er aber nicht auf ein Element eines „zehnelementigen Vektors von beliebig großen Vektoren von irgendwas" zugreifen, denn er weiß ohne weiteres nicht, wie groß ein „double[j]" ist und kann folglich mit Vektoren dieses Typs nichts anfangen.

Wie löscht man nun einen solchen Vektor variabler Größe wieder? Die Laufzeitbibliothek des C++-Systems merkt sich beim Allozieren eines Vektors variabler Größe irgendwo, wie groß der nun wirklich war. Beim Löschen ist dann hinter das „delete" ein Klammerpaar „[ ]" zu setzen:

```
void main()
{ int anz;

  unsigned *varivec = new unsigned[anz]; // ...

  delete [ ] varivec;
}
```

Bei älteren C++-Versionen („1.0-Standard") mußte man in diesen Klammern sogar die Vektorgröße angeben:

```
delete [anz] varivec;
```

Das ist heutzutage nicht mehr nötig. MaxonC++ akzeptiert es, um die Kompatibilität zu den alten Compilern zu wahren. Es ist absolut egal, was man in diese Klammern schreibt. Weglassen darf man diese Klammern aber nicht, sonst steigt das Programm wieder einmal aus.

2.6.4 Initialisierungen

Jede Variable kann initialisiert werden, folglich ist es nur recht und billig, wenn man dynamisch erzeugte Objekte ebenfalls initialisieren kann. Dabei ist der Initialisierungsausdruck beim „new" hinter den Datentyp zu setzen:

```
void fun(int *ip)
{ ip = new int(42);
}
```

Die Allozierungsanweisung ist also identisch mit

```
ip = new int;
*ip = 42;
```

Es gibt dabei allerdings ein paar Einschränkungen:

- Referenzen können nicht erzeugt und folglich auch nicht initialisiert werden.
- Vektoren können nicht initialisiert werden. Das kommt wohl daher, daß die Vektorgröße bei „new" variabel sein kann, C++ sonst aber nur Vektoren bekannter konstanter Größe initialisieren kann. Man müßte für diesen Fall völlig neue Regeln erfinden und implementieren, und das lohnt den Aufwand einfach nicht. Mit „new" erzeugte Vektoren fester Größe könnten zwar prinzipiell nach den bekannten Regeln initialisiert werden, aber so etwas kommt in der Praxis kaum vor (Sie erinnern sich, satt einen Vektor fester Größe zu allozieren, kann man ihn genausogut ganz normal deklarieren) und deshalb hat man darauf verzichtet.
- Strukturen können nicht mit Elementlisten initialisiert werden:

```
struct S { int a, b; };

S *sp = new S({1, 2});  // SYNTAX ERROR
```

Das erscheint auf den ersten Blick als ein großes Manko, aber Elementlisten sind ein altes, konventionelles C-Feature, während man Strukturen in C++ in der Regel mit „Konstruktoren" initialisieren wird. Da „new" ein neues Feature von C++ ist, hatte man wohl keine rechte Lust, hier auch noch diese antiquierten Elementlisten zu unterstützen.

Da ich gerade dabei bin, zu den objektorientierten Features vorzugreifen, will ich hier noch erwähnen,

- daß man beim Erzeugen einer Struktur mit einem „Konstruktor" stets geeignete Argumente dafür als Initialisierung angeben muß, es sei denn, die Struktur hat einen „Default-Konstruktor",

- daß ein Vektor von Strukturen nur dann mit „new" alloziert werden kann, wenn die Struktur einen „Default-Konstruktor" besitzt, welcher dann für jedes einzelne Vektorelement aufgerufen wird, und

- daß beim Löschen einer Struktur mit „delete" ggf. der „Destruktor" dieser Struktur aufgerufen wird. Entsprechend wird bei Vektoren von Strukturen der Destruktor für jedes einzelne Vektorelement aufgerufen.

Aber von all' diesen geheimnisvollen Features gehen wir jetzt zuerst noch einmal zurück in das finstere C-Mittelalter:

2.6.5 „malloc" und andere Antiquitäten

Wie bereits erwähnt, hatten in alten Tagen die C-Programmierer (die es heute angeblich immer noch geben soll, aber das ist bestimmt nur eine Legende) ein hartes Leben: ANSI C kennt weder „new" noch „delete". Statt dessen lief die Speicherverwaltung im wesentlichen über zwei Funktionen aus „<stdlib.h>". Man reservierte sich Speicher mit der Funktion

```
void *malloc(size_t size);
```

Dabei bezeichnet „size_t" einen Datentypen, der ebenfalls in „<stdlib.h>" deklariert wird und gerade der Typ ist, den „sizeof" als Ergebnis hat (in MaxonC++ ist das „unsigned int"). Die Funktion „malloc" reserviert jedenfalls „size" Bytes und gibt einen Zeiger darauf zurück:

```
#pragma -
#include <stdlib.h>

void main()
{ int *ip = malloc(sizeof(int));
  /* usw. */
}
```

Man beachte, daß man in C++ nicht so ohne weiteres ein „void*“ in einen anderen Pointertyp verwandeln kann, weshalb man in C++ einen Cast setzen müßte:

```
int *ip = (int*) malloc(sizeof(int));
```

Aber es wird wohl niemand mehr in C++ noch „malloc“ verwenden.

Man kann im allgemeinen nicht erwarten, daß ein „n“-elementiger Vektor gerade „n“-mal so groß ist wie sein Elementtyp. Die Struktur

```
struct S { int i; char c; };
```

z. B. nimmt in MaxonC++ gerade fünf Bytes ein, nämlich vier für „int“ und eins für „char“. Auf dem MC68000 und vielen anderen Prozessoren auch müssen Langwortdaten aber immer an einer geraden Adresse stehen. Deshalb wird der Compiler bei einem Vektor von „S“ für jedes Element ein zusätzliches Füllbyte reservieren, so daß jedes Vektorelement und damit auch das darin enthaltene „i“ an einer geraden Adresse liegt. MaxonC++ fügt hier sogar drei Bytes ein, um eine Langwort-Ausrichtung zu erzwingen. Langer Rede, schwacher Sinn: Es ist nicht empfehlenswert, mit „malloc“ einen Vektor variabler Größe einzurichten, denn

```
malloc(n * sizeof(struct S))
```

führt aus den oben angeführten Gründen oft zu falschen Ergebnissen, und

```
malloc(sizeof(struct S[n]))
```

ist in C ganz einfach falsch, denn hier gibt es keine Vektoren nicht-konstanter Größe. Also gibt es auch noch die Funktion

```
void *calloc(size_t nobj, size_t size)
```

zum korrekten Allozieren von Vektoren, wobei der erste Parameter die Anzahl der Elemente und der zweite die normale Größe des Elementtyps darstellt:

```
struct S *var_s_vec = calloc(n, sizeof(struct S))
```

„calloc“ kennt die „Alignment“-Regeln des jeweiligen C-Systems und kann deshalb die notwendige Anzahl von Füllbytes einfügen. „malloc“ steht wohl für „Memory ALLOCation“, aber was das „c“ am Anfang von „calloc“ heißen soll, weiß ich auch nicht. Mir persönlich würde der Name „colloc“ viel besser gefallen, denn das ist wenigstens ein Palindrom.

Zu guter Letzt braucht man natürlich auch noch eine Funktion, die Speicher wieder freigibt:

```
void free(void *p);
```

Man übergibt hier wie bei der „delete"-Anweisung einen Zeiger auf das zu löschende Objekt, das entweder mit „malloc" oder „calloc" erzeugt worden sein muß, z. B.

```
free(var_s_vec);
```

Man hat hier also keine unterschiedlichen Freigabeanweisungen wie in C++ („delete" und „delete[]").

Man sieht sofort, daß diese C-Funktionsaufrufe nicht gerade besonders schön aussehen, und da man in C++ ohnehin dabei war, C gründlich zu erweitern, hat man die viel schöneren „new"- und „delete"-Ausdrücke eingeführt. „new" und „delete" sind übrigens zwei Wortsymbole und damit ein echter Bestandteil der Programmiersprache C++, während „malloc" und „free" keine wirklichen C-Sprachkonstrukte, sondern nur zwei Bibliotheksfunktionen wie andere auch sind. Das Konzept der dynamischen Datenorganisation wurde in C++ also „aufgewertet".

Neben diesen ästhetischen Überlegungen gibt es aber auch noch ganz handfeste Gründe, weshalb „malloc" und „free" für objektorientierte Programmierung ungeeignet sind: Der C++-Compiler muß aus Gründen der Konsistenz eine Möglichkeit haben, festzustellen, ob ein neues Datenobjekt erzeugt bzw. ein bestehendes gelöscht wird, weil dabei eventuell Konstruktoren oder Destruktoren aufgerufen werden müssen.

Für alle, die immer noch am sittlichen Nährwert der dynamischen Datenorganisation zweifeln, und insbesondere alle die, die glauben, daß sie so etwas nie kapieren werden, möchte ich zu diesem Thema ein etwas größeres Beispiel bringen:

2.6.6 Beispielprogramm: Eine Liste von Daten

2.6.6.1 Das Problem: Ein wirrer Haufen von Daten

Ein Problem, auf das man als Programmierer immer wieder stößt, sind Listen von irgendwelchen Daten. Anfänger, lösen so etwas mit Vektoren, was zwar einfach und auch schnell ist, aber den ganz gewaltigen Nachteil hat, daß man normalerweise Vektoren fester Größe deklarieren muß. Auch Vektoren variabler Größe (siehe 2.6.3) sind nicht so ganz das Gelbe vom Ei, denn hier muß man schon in dem Moment, in dem die Datenstruktur erzeugt wird, wissen, wie viele Elemente die Liste höchstens umfassen soll. Nein, für echte Programmierer führt wirklich kein Weg an dynamischen Listen vorbei.

Die lineare Liste ist bestimmt nicht die beste, aber die einfachste und deshalb gebräuchlichste Datenstruktur. Auch das Amiga-Betriebssystem macht intern von solchen Strukturen ausgiebig Gebrauch. Was versteht man aber unter einer Liste?

Als Beispiel wollen wir ein kleines Telefonbuch erzeugen. Jeder Eintrag umfaßt hier zunächst einen Namen und eine Telefonnummer:

```
struct Eintrag_erster_Entwurf
{ char Name[30];
  char Nummer[20];
};
```

Nun stellen wir an unsere Datenstruktur aber folgende Anforderungen:

1. Die Anzahl der Datensätze ist ausschließlich durch das zur Verfügung stehende RAM begrenzt. Dabei soll die Struktur zu keinem Zeitpunkt mehr Speicher als unbedingt erforderlich benötigen.

2. Die Datensätze können in beliebiger Reihenfolge eingegeben werden und werden dabei sofort sortiert.

3. Daten können aus der Struktur gelöscht werden. Dabei können Einfüge- und Löschoperationen in beliebiger Reihenfolge erfolgen.

Und alles das soll auch noch halbwegs schnell vor sich gehen. Wie packt man nun ein solches Problem an?

Es führt offensichtlich kein Weg daran vorbei, die Daten dynamisch im RAM einzurichten. Die Idee der Liste ist, daß jedes Element einen Zeiger auf das nachfolgende besitzt, etwa so:

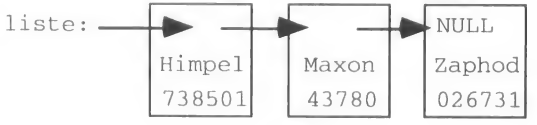
```
struct Eintrag_erster_Entwurf
{ Eintrag_erster_Entwurf *next;
  char name[30];
  char nummer[20];
};
```

Alles, was wir jetzt noch brauchen, ist ein Zeiger auf das erste Listenelement:

```
Eintrag_erster_Entwurf *liste;
```

Der Name des ersten Listenelements ist dann wie gewohnt „liste->name", der des zweiten Elements „liste->next->name", an das dritte Listenelement gelangt man über „liste->next->next->name" und so weiter. Natürlich muß jede Liste ein Ende haben, und

das markiert man normalerweise, indem man dort als „Next"-Zeiger eine Null einträgt. Ein „Telefonbuch" mit drei Einträgen könnte also folgendermaßen aussehen:



Es dürfte anschaulich klar sein, wie man sich mit einem Zeiger durch eine solche Struktur hangeln kann, aber nur in eine Richtung, vom Anfang zum Ende. Man nennt eine solche Struktur eine „einfach verkettete Liste".

2.6.6.2 Die Lösung: Eine doppelt verkettete Liste

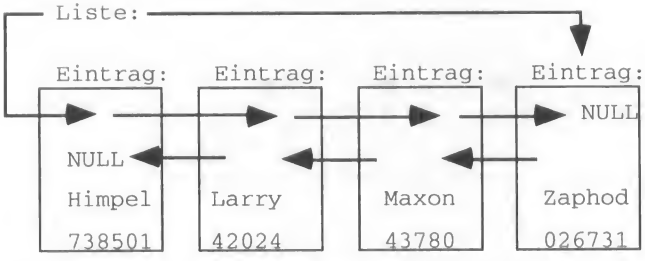
Etwas komplizierter ist eine doppelt verbundene Liste, mit der wir uns im folgenden befassen wollen. Dabei hat jedes Listenelement gleich zwei Zeiger, wie gehabt einen auf den Nachfolger und zusätzlich einen auf den Vorgänger:

```
struct Eintrag
{ Eintrag *next, *prev;

  char name[30];
  char nummer[20];
};
```

Auch jetzt reicht es eigentlich, wenn wir einen Zeiger auf den Listenanfang haben, aber da wir schon die Möglichkeit haben, rückwärts durch die Struktur zu laufen, benutzen wir hier aus Symmetriegründen lieber gleich zwei Zeiger: Einen auf das erste und einen auf das letzte Element der Liste. Das schreit wiederum danach, diese beiden Zeiger in einer neuen Struktur namens „Liste" zusammenzufassen:

```
struct Liste
{ Eintrag *anfang, *ende;
};
```

Am besten macht man sich auch das wieder mit einem „Bildchen" klar, diesmal gleich mit vier Einträgen:



Das sieht ja schon ganz schön kompliziert aus, und wir haben es nun auch mit zwei verschiedenen Datentypen zu tun. Aber keine Angst, wenn man sich erst einmal an das Listen-Konzept gewöhnt hat, ist das ganz einfach - ehrlich!

Zum „Aufwärmen" schreiben wir uns eine Funktion, die alle Daten einer Liste ausgibt. Wir nennen sie folgerichtig „Ausgabe" und als Parameter erhält sie eine Referenz auf eine „Liste". Wir benutzen hier besser eine Referenz, weil dann beim Funktionsaufruf nur ein Zeiger auf die Liste auf den Stack kopiert werden muß, während sonst die ganze Struktur (immerhin zwei Zeiger) kopiert werden müßte. Weil wir aber nicht vorhaben, bei der Ausgabe irgend etwas an der Liste zu verändern, benutzen wir die Qualifizierung „const":

```
void Ausgabe (const Liste &L)
{
```

Wie geht's jetzt weiter? Wir müssen irgendwie durch die ganze Liste laufen, und zwar vom Anfang bis zum Ende. Also brauchen wir einen Zeiger auf ein „Element", den wir gleich auf das erste Element der Liste setzen:

```
  Eintrag *E = L.anfang;
```

Jetzt müssen wir folgende drei Schritte iterieren:

1. Prüfe, ob das Listenende erreicht ist; falls ja, sind wir fertig.
2. Andernfalls gebe das Listenelement, auf das „E" zeigt, aus.
3. Setze „E" auf das nächste Listenelement und mach bei (1.) weiter.

Genau das besorgt die folgende Schleife:

```
while (E != 0)
{ cout « E->name « ": " « E->nummer « "\n";
  E = E->next;
}
```

Hier gibt es noch einiges zu verbessern, aber wir wollen es einmal dabei belassen. Die komplette Funktion sieht also so aus:

```
#include <stream.h>

void Ausgabe(const Liste &L)
{ Eintrag *E = L.anfang;

  while (E != 0)
  { cout « E->name « ": " « E->nummer « "\n";
    E = E->next;
  }
}
```

Man könnte sie aber auch so schreiben:

```
void Ausgabe (const Liste &L)
{ for(Eintrag *E = L.anfang; E; E = E->next)
  cout « E->name « ": " « E->nummer « "\n";
}
```

Durch scharfes Hinsehen werden Sie feststellen, daß es sich hier nur um eine andere Schreibweise desselben Algorithmus handelt. C ist eben eine sehr „flexible" Sprache - man kann Programme beliebig konfus schreiben. „But now to something completely different:"

2.6.6.3 Initialisieren und Einfügen

Auch eine Liste muß initialisiert werden, nämlich als leere Liste. Am besten schreibt man sich auch dafür eine kleine Funktion:

```
void Init(Liste &L)
{ L.anfang = 0;
  L.ende   = 0;
}
```

Das Schöne an C++ ist ja, daß man Funktionen überladen kann. Sie können also in einem Programm -zig Datentypen haben und für jeden so eine Funktion namens „Init" definieren, der Compiler weiß, welche er zu nehmen hat. Noch nicht einmal das müssen Sie tun, denn dafür gibt es Konstruktoren und Destruktoren, aber das gehört jetzt nicht hier hin.

Eine leere Liste ist nicht besonders aufregend, also brauchen wir dringend eine Funktion, die einen Datensatz in die Liste einfügt. Auch dafür schreiben wir eine Funktion, die als Parameter eine Listenstruktur, einen Namen und eine Telefonnummer erhält. Die

Funktion liefert einen Zeiger auf das neu erzeugte Listenelement zurück. Wenn kein Speicher reserviert werden konnte, gibt sie eine Null zurück.

Also deklarieren wir unsere Funktion - aber nicht, ohne vorher die Stringfunktionen einzubinden:

```
#include <string.h>

Eintrag *Einfueg(Liste &L, const char Name[], const char
Nummer[])
{
```

Es geht jetzt ziemlich simpel los: Wir erzeugen ein neues Element und geben eine Null zurück, wenn das fehlschlug:

```
Eintrag *neu = new Eintrag;

if (!neu)
  return 0;
```

Jetzt sollten wir das neue Element noch mit den Argumenten der Funktion initialisieren, wobei wir wegen dieses von C geerbten *%&@ß-Stringhandlings auf die „strcpy"-Funktion zurückgreifen müssen:

```
strcpy(neu->name, Name);
strcpy(neu->nummer, Nummer);
```

So, nun wird es kompliziert. Als erstes müssen wir herausfinden, wo wir das neue Element in der Liste einhängen sollen. Deshalb lassen wir einen Zeiger namens „pos" durch die Liste laufen, bis wir ein Element finden, dessen „name" alphabetisch größer als der neue ist, wobei allerdings der Sonderfall auftreten kann, daß es kein solches Element gibt:

```
Eintrag *pos = L.anfang;

while (pos!=0 && strcmp(pos->name, Name) < 0)
  pos = pos->next;
```

Man beachte, daß C logische Ausdrücke grundsätzlich minimal auswertet: Wenn die Klausel „pos != 0" schon nicht erfüllt ist, wird der zweite Teil der Schleifenbedingung, nämlich das „strcmp", gar nicht mehr ausgewertet.

Was heißt das nun, wenn die Schleife mit „pos == 0" anhält? Offensichtlich ist das neue Element das größte der ganzen Liste, was aber auch daran liegen kann, daß die Liste ganz einfach leer ist. Also testen wir zuerst diese Sonderfälle ab:

```
if (pos == 0)
// Sonderfall: Anfügen an das Ende der Liste
{ Eintrag *alt = L.ende;// Altes Listenende
```

Der Zeiger „alt" zeigt also auf das bisher letzte Element der Liste. Wenn die Liste bisher leer war, ist „alt" natürlich Null, andernfalls:

```
if (alt != 0)
// Allgemeiner Unter-Fall: wirklich anhängen
{ alt->next = neu;
  neu->prev = alt;
  neu->next = 0;
  L.ende = neu;
}
```

Langsam zum Mitdenken: Zuerst setzen wir „neu", also das neue Listenelement, als Nachfolger von „alt", also dem bisherigen Listenende, und anschließend definieren wir umgekehrt „alt" als Vorgänger von „neu". Nun müssen wir nur noch den Nachfolgerzeiger des neuen Elements auf Null setzen, denn es ist nun ja das letzte Listenelement, und den Eintrag „ende" der Listenstruktur aktualisieren, indem wir ihn auf das neue Element umhängen. Wenn Sie das jetzt nicht verstanden haben, nehmen Sie sich am besten einen Bleistift und die Rückseite eines gebrauchten Briefumschlags und malen sich die ganze Sache auf.

Es fehlt noch das „else" zum zweiten „if", also der Fall, daß sowohl „pos" als auch „alt" sind Null. Das kann nur daran liegen, daß unsere Liste noch völlig leer ist:

```
else
// Spezieller Spezialfall: Liste ist noch leer
{ neu->next = 0;
  neu->prev = 0;
  L.anfang = neu;
  L.ende = neu;
}
```

Diese Zuweisungen dürften halbwegs verständlich sein: Das neue Element ist das einzige und hat deshalb weder Vorgänger noch Nachfolger. Es ist jetzt gleichzeitig das erste und das letzte Listenelement, weshalb beide Zeiger der Listenstruktur auf das neue Element gesetzt werden müssen.

Nun poppen wir eine Ebene 'rauf und überlegen uns, was wir tun, wenn „pos" nicht auf Null zeigt, wir also ein Element gefunden haben, vor das wir das neue hängen müssen. Hier gibt es dann gleich noch einen Sonderfall: Möglicherweise ist das neue Element das kleinste der ganzen Liste und muß deshalb an den Anfage gehängt werden. Das erkennt man wahlweise daran, daß „pos == L.anfang" gilt oder daß „pos" keinen Vorgänger hat:

```
  }
  else
  if (pos->prev == 0)
  // Sonderfall: Neues Element an Listenanfang hängen
  { pos->prev = neu;
    neu->next = pos;
    neu->prev = 0;
    L.anfang = neu;
  }
```

Das geht völlig analog zum Fall „Anhängen an das Listenende", nur daß wir hier mit „pos" schon einen Zeiger auf das bisher erste Listenelement haben und deshalb keine Hilfsvariable wie „alt" einführen müssen. Nun kommen wir endlich zum ganz gewöhnlichen, allgemeinen Fall, daß das neue Element zwischen zwei andere gehängt werden muß:

```
  else
  // Allgemeiner Fall: Element in Liste einhängen
  { Eintrag *vor = pos->prev;
```

Die Hilfsvariable „vor" wird uns einige Schreibarbeit ersparen, echte Pointerartisten könnten aber auch ohne sie auskommen. Das neue Element muß jetzt zwischen die Elemente, auf die „vor" bzw. „pos" zeigen, eingehängt werden. Zuerst hängen wir es hinter „vor":

```
    vor->next = neu;
    neu->prev = vor;
```

...und anschließend vor „pos":

```
    neu->next = pos;
    pos->prev = neu;
  }
```

Damit wären wir fertig und müssen nur noch am Ende der Funktion einen Zeiger auf das neue Element zurückgeben:

```
  return neu;
  }
```

Gar nicht so einfach - oder? Ich hoffe, daß ich Ihnen damit nicht ein für allemal die Lust auf dynamische Speicherorganisation genommen habe. Es gibt auch eine wesentlich einfachere Datenstruktur, nämlich die einfach verkettete Liste, und Varianten unseres Konzepts, bei denen man gleich bei der Initialisierung der Listenstruktur „Phantom-Elemente" an den Anfang bzw. das Ende der Liste hängt, was einem die vielen

Sonderfälle erspart. Aber wir bleiben unbeirrt bei unserem Listenkonzept und schreiben dafür jetzt noch zwei unverzichtbare Routinen:

2.6.6.4 Suchen und Löschen

MaxonC++ löscht zwar am Programmende automatisch alle dynamischen Objekte, aber trotzdem ist es sinnvoll, wenn man sich eine Routine schreibt, die eine ganze Datenstruktur löscht. Hier ist sie:

```
void AllesLoeschen(Liste &L)
{ Eintrag *p = L.anfang;

  while(p)
  { Eintrag *hilf = p;
    p=p->next;
    delete hilf;
  }
  Init(L);
}
```

In der Schleife wird wieder die Liste durchlaufen, wobei alle Einträge gelöscht werden. Anschließend wird der Ordnung halber „Init" aufgerufen, wodurch die beiden Zeiger der Listenstruktur wieder auf Null gesetzt werden.

Wieso braucht man hier den Hilfszeiger „hilf"? In der Schleife sind zwei Schritte auszuführen: Das Element, auf das „p" gerade zeigt, ist zu löschen, und „p" ist auf das nachfolgende Element zu setzen. Wenn man erst „p" weitersetzt, löscht man natürlich das falsche Element, aber wenn man erst „delete" aufruft, liest man anschließend bei „p = p->next" aus einem Datenelement, das man ja soeben gelöscht hat - eine etwas haarige Sache! Also gibt es nur eine Lösung: Man setzt erst einen Hilfszeiger auf das aktuelle Lietenelement, geht dann mit „p" ein Element weiter und löscht das fragliche Listenelement dann über den Hilfszeiger.

Wir wollen aber nicht immer die ganze Liste löschen, sondern auch einzelne Elemente aus der Liste entfernen. Dafür dient die folgende Funktion:

```
void Loeschen(Liste &L, Eintrag *E)
{
```

Ein Blick auf die Listenstruktur sagt uns, daß wir vor dem Löschen des Elements „E" dessen Vorgänger und Nachfolger verändern müssen:

```
  Eintrag *vor = E->prev, *nach = E->next;
```

Es kann natürlich sein, daß „E“ gar keinen Vorgänger hat, dann handelt es sich offenbar um den Listenanfang. Andernfalls ist der neue Nachfolger des Vorgängers der bisherige Nachfolger des zu löschenden Elements (welch ein Satz!):

```
if (vor)
  vor->next = nach;
```

Falls wir es mit dem Listenanfang zu tun haben, müssen wir zwar nicht dessen Vorgänger (den es ja nicht gibt), wohl aber die Listenstruktur selbst verändern:

```
else
  L.anfang = nach;
```

Dadurch wird der Nachfolger des bisher ersten Elements der neue Listenanfang. Auf der anderen Seite des zu löschenden Elements läuft alles ganz analog:

```
if (nach)
  // Element hat Nachfolger: Dann dessen Vorgänger verändern
  nach->prev = vor;
else
  // Kein Nachfolger, also Listenende anpassen
  L.ende = vor;
```

Nun haben wir den Eintrag aus der Liste ausgehängt und geben last not least seinen Speicher wieder frei:

```
  delete(E);
}
```

Um diese Löschfunktion zu benutzen, braucht man einen Zeiger auf ein Element, aber wie bekommt man den? Nun, zum Beispiel indem man mit der folgenden Funktion einen Eintrag aufgrund seines Namens sucht:

```
Eintrag *Suche(const Liste &L, char *Name)
{ for( Eintrag *p = L.anfang; p && strcmp(p->name, Name);
  p = p->next );
  return p;
}
```

Verglichen mit den anderen, ist diese Funktion wirklich nicht sehr kompliziert. Deshalb habe ich mir die Freiheit genommen, sie etwas konfus aufzuschreiben: Anstelle der „while“-Schleifen, die in den anderen Funktionen zum Durchlaufen der Liste benutzt wurde, nehmen wir hier eine „for“-Schleife. Vor der Schleifenausführung wird der Zeiger „p“ mit dem Listenanfang initialisiert (wobei Sie sich nicht wundern sollten, daß „p“ dabei deklariert wird, denn das ist in C++ erlaubt), dann folgt auch schon die

Schleifenbedingung, die aus zwei Klauseln besteht. Als „echter" C-Hacker habe ich jedesmal das „!= 0" weggelassen, denn das ist bekannlich zwar übersichtlicher, aber redundant. Die Schleifenbedingung könnte man also auch

```
p != 0 && strcmp(p->name, Name) != 0
```

schreiben. Der dritte Teil jeder „for"-Schleife ist die Anweisung, die nach jedem Durchlaufen der Schleife ausgeführt werden soll. Hier setzt sie in bekannter Weise den Zeiger „p" weiter.

Und was ist mit dem Schleifenrumpf? Nun, wir tun doch schon alles, was nötig ist, deshalb entfällt er hier ersatzlos, und er wird durch ein schlichtes, einsames Semikolon ersetzt. Am Ende der Funktion müssen wir nur noch den Wert von „p" zurückgeben, denn dieser zeigt dann entweder auf die gesuchte Person oder ist Null, wenn der Name nicht gefunden wurde. Dies ist wieder ein nettes Beispiel dafür, daß man in C ausgesprochen kompakte Programme schreiben kann, die dann nicht gerade gut lesbar sind.

2.6.6.5 Das Programm an einem Stück

Aus Gründen der besseren Übersicht folgt jetzt noch einmal das Ganze Listenprogramm. Außerdem braucht jedes Programm ein Hauptprogramm, weshalb ich noch eine kleine Funktion „main" angehängt habe.

```
/*
Doppelt verkettete Listen

Ein kleines Beispiel für dynamische Datenstrukturen in
C++, aber ohne objektorientierte Programmierung

Programmiert von Jens Gelhar am 13.01.92
*/

#include <stream.h>
#include <string.h>

// * * * Datentypen * * *

struct Eintrag     // Ein einzelner "Telefonbuch"-Eintrag
{ Eintrag *next, *prev;   // Vorgänger und Nachfolger

  char name[30];      // Name und...
  char nummer[20];    // Telefonnummer
};
```

```
struct Liste      // Sortierte Liste von Einträgen
{ Eintrag *anfang, *ende;   // Erstes bzw. letztes Listenelement
};

// * * * Funktionen * * *

void Init(Liste &L)  // Initialisiert eine Listenstruktur,
                     // indem sie ihre beiden
                     // Pointer auf 0 setzt:
{ L.anfang = 0;
  L.ende   = 0;
}

void Ausgabe (const Liste &L) // Telefonbuch ausgeben
{ for(Eintrag *E = L.anfang; E; E = E->next)
   cout « E->name « ": " « E->nummer « "\n";
}

Eintrag *Einfueg(Liste &L, const char Name[], const char Nummer[])
  // Fügt einen neuen Eintrag in die Liste ein
{ // Neues Element erzeugen:
  Eintrag *neu = new Eintrag;

  if (!neu) return 0;     // Kein Speicher frei!

  // Neues Element initialisieren:
  strcpy(neu->name, Name);
  strcpy(neu->nummer, Nummer);

  // Richtige Position zun Einfügen suchen:
  Eintrag *pos = L.anfang;

  while (pos!=0 && strcmp(pos->name, Name) < 0)
    pos = pos->next;

  // Element in Liste einhängen, dabei diverse
  // Sonderfälle beachten:

  if (pos == 0) // Sonderfall: Anfügen an das Ende der Liste
  { Eintrag *alt = L.ende;// Altes Listenende

    if (alt != 0) // Allgemeiner Unter-Fall: wirklich anhängen
    { alt->next = neu;
      neu->prev = alt;
      neu->next = 0;
      L.ende = neu;
    }
```

```
    else      // Spezieller Spezialfall: Liste ist noch leer
    { neu->next = 0;
      neu->prev = 0;
      L.anfang = neu;
      L.ende = neu;
    }
  }
  else
    if (pos->prev == 0)  // Sonderfall: Neues Element an
                         // Listenanfang hängen
    { pos->prev = neu;
      neu->next = pos;
      neu->prev = 0;
      L.anfang = neu;
    }
    else
    // Allgemeiner Fall: Element in Liste einhängen
    { Eintrag *vor = pos->prev;
      // „neu" zwischen „vor" und „pos" einhängen:
      vor->next = neu;
      neu->prev = vor;
      neu->next = pos;
      pos->prev = neu;
    }

  // Fertig, Zeiger auf neues Element zurückgeben:
  return neu;

}

void AllesLoeschen(Liste &L)
// Löscht sämtliche Listenelemente
{ Eintrag *p = L.anfang;

  while(p)
  { Eintrag *hilf = p;
    p=p->next;
    delete hilf;
  }
  Init(L);
}
```

```
void Loeschen(Liste &L, Eintrag *E)
  // Löscht das Element, auf das „E" zeigt, aus Liste „L"
{ // Vorgänger unf Nachfolger des zu löschenden Elements
  Eintrag *vor = E->prev, *nach = E->next;

  if (vor)  // Element hat Vorgänger: dann dessen
            // Nachfolger umsetzen
    vor->next = nach;
  else      // Kein Vorgänger: dann Listenanfang aktualisieren
    L.anfang = nach;

  if (nach) // Element hat Nachfolger: dann dessen
Vorgänger
            // verändern
    nach->prev = vor;
  else      // Kein Nachfolger, also Listenende anpassen
    L.ende = vor;

  delete(E);
}

Eintrag *Suche(const Liste &L, char *Name)
// Sucht einen Eintrag mit diesem Namen und
// gibt Zeiger auf das
// zugehörige Listenelement zurück, oder 0 bei Fehler
{ for( Eintrag *p = L.anfang; p && strcmp(p->name, Name);
    p = p->next );
  return p;
}

// * * *  Hauptprogram  * * *

void main()
{ // Eine Liste wird deklariert und initialisiert:

  Liste l;
  Init(l);

  // Ein paar Beispieldaten werden eingefügt:

  Einfueg(l, "Harley-Davidson", "001-414/342-4680");
  Einfueg(l, "Boris Becker", "0815/4711");
  Einfueg(l, "James Bond", "007/26731");
  Einfueg(l, "Maxon", "06196/481811");
  Einfueg(l, "Zaphod Beeblebrox", "00042/08154242");
```

```
  Einfueg(l, "Luxemburg", "00352");
  Einfueg(l, "Queensr\0377che", "795069");
  Einfueg(l, "Fishbone", "4676152");

  // Einen Datensatz suchen und ggf. löschen:

  Eintrag *e = Suche(l,"Maxon");
  if (e)
    Loeschen(l, e);
  else
    cout « "Name nicht gefunden!\n";

  // Restliche Liste ausgeben:
  Ausgabe(l);

  // ...und wieder aufräumen:
  AllesLoeschen(l);

}
```

3. Objektorientiertes Programmieren

3.1 Klassen und Objekte

3.1.1 Keine Panik!

Viele Leute behaupten, das Konzept der objektorientierten Programmierung werde die Softwareentwicklung revolutionieren, weit stärker als das etwa die Einführung der strukturierten Programmiersprachen es in den beiden vergangenen Jahrzehnten getan hat, und wahrscheinlich werden diese Leute recht behalten.

Die Sache hat bisher einen kleinen Haken: So mancher, der von „objektorienterter Programmierung" spricht, hat keine rechte Ahnung davon, was das eigentlich ist. Auch viele erfahrene Programmierer stehen ratlos vor Begriffen wie „Klasse", „Methode" oder „Vererbung" und verstehen die Welt nicht mehr. Was ist geschehen? Wird jetzt alles anders? Sind fünf Jahrzehnte prozeduraler Programmierung plötzlich nichts mehr wert?

Aber die Rettung naht, ist sogar schon da: Kapitel 3 des Handbuchs zu MaxonC++, jawohl, genau das, was Sie jetzt gerade lesen!

Zunächst sollten Sie das bekannte Motto, das in großen, freundlichen Buchstaben über diesem Abschnitt steht, beherzigen. Damit das Ihnen etwas leichter fällt, werde ich im folgenden versuchen, halbwegs klarzustellen, was es mit der objektorientierten Programmierung eigentlich auf sich hat.

Das Problem dabei, das auch zu der allgemeinen Verwirrung und Unkenntnis im Zusammenhang mit OOP („objekt-orientierter Programmierung") erheblich beiträgt, ist, daß das Ganze nur bei großen, komplexen Projekten Sinn macht. Es ist zwar durchaus möglich, ein objektorientiertes „Hello World"-Programm zu schreiben, aber einen sittlichen Nährwert hat das nicht. Aber nehmen wir als Beispiel doch ganz einfach das Listen-Programm, das am Ende von Kapitel zwei ausführlich dargestellt wurde. Darin könnte man nämlich schon einige (mit etwas Mühe sogar alle) objektorientierten Features sinnvoll einsetzen.

Eine Grundidee der objektorientierten Programmierung ist, den Daten des Programms eine gewisse Intelligenz zu verleihen. Bei größeren Programmen stellt man schnell fest, daß man eigentlich herzlich wenig mit reinen Algorithmen zu tun hat, im wesentlichen bestehen viele Programme aus Datenstrukturen und Funktionen, die auf diesen Daten operieren. Oft liegt die ganze Cleverness beim Programmieren im geschickten Design der Datenstrukturen, während die Algorithmen des Programms vergleichsweise trivial sind. Und damit wären wir auch schon bei einigen im Zusammenhang mit OOP vielbenutzten Begriffen:

Eine „Klasse" ist zunächst nichts anderes als ein ganz gewöhnlicher Datentyp, wie Sie ihn aus der „konventionellen" (d. h. nicht objektorientierten) Programmierung kennen, nur daß zu einer Klasse in der OOP zusätzlich noch Funktionen gehören können. Die Daten und die darauf definierten Funktionen einer Klasse nennt man zusammen sehr vornehm auch die „Eigenschaften" der Klasse.

In C++ ist das grundlegende Sprachkonstrukt zur Definition von Klassen die Struktur, die es ja auch schon in C gibt. Nun kann man in C++ als Member einer Struktur nicht nur Daten, sondern auch Funktionen deklarieren. Diese Funktionen werden dadurch formal an die Struktur, oder sagen wir lieber Klasse, angebunden. Zum Beispiel haben wir im Telefonlisten-Programm zuerst den Datentyp „Liste" und anschließend eine Funktion namens „Einfueg" definiert, die dann ziemlich beziehungslos im Programmtext 'rumlagen. In C++ kann man die Funktion „Einfueg" direkt in die Klassendefinition hineinschreiben, so daß sie Bestandteil der Klasse und ihrer Eigenschaften wird. Solche Funktionen bezeichnet man auch als „Methoden", aber ich persönlich mag diesen Begriff nicht und ziehe die Bezeichnung „Member-Funktion" vor, zumal auch Stroustrup die schlicht Bezeichnung „Member function" bevorzugt.

Man erhält nun eine klare Trennung zwischen den Funktionen, die zu einer Klasse gehören, und allen anderen, die nur „von außen" auf die Klasse zugreifen und die Klasse und ihre Eigenschaften lediglich benutzen. Der Gedanke liegt nahe, den beiden Funktionstypen unterschiedliche Rechte einzuräumen. In C++ dürfen die Member-Funktionen stets „alles" mit einem Klassenobjekt tun, während der Programmierer festlegen kann, welche Eigenschaften der Klasse „der Allgemeinheit" zur Verfügung stehen. Auch das läßt sich wieder an unserer Klasse „Liste" veranschaulichen: Hier bietet es sich an, die beiden Datenmember „anfang" und „ende" als „privat" zu deklarieren, so daß man diese von außen nur indirekt über Funktionen wie „Init", „Einfueg" usw. benutzen kann. Dadurch ist sichergestellt, daß kein Programmteil versehentlich an diesen beiden Zeigern herumspielen kann. Auch bei der Klasse bzw. Struktur „Eintrag" würde man am besten die Member „prev" und „next" als privat deklarieren, denn diese sind für den „Benutzer" der Klasse nicht von Interesse, während man „name" und „nummer" eher als öffentlich deklarieren würde. C++ bietet einige durchdachte Features, um derartiges „Information Hiding" zu realisieren und so zwischen den internen Daten und der öffentlichen „Schnittstelle" einer Klasse zu unterscheiden, was als „Separation of Concerns" („Trennung von Angelegenheiten") bezeichnet wird.

Am Telefonlisten-Programm wird Ihnen vielleicht aufgefallen sein, daß man ein Objekt des Typs „Liste" erst mit der Funktion „Init" initialisieren muß, bevor man es benutzen kann, und daß man die Funktion „AllesLoeschen" aufrufen muß, wenn man die Liste nicht mehr benötigt. In C++ und anderen OO-Programmiersprachen stehen Ihnen Features zur Verfügung, die derartige Vorgänge automatisieren, nämlich „Konstruktoren" und „Destruktoren". Man deklariert einfach einen Konstuktor, der ein „Liste"-Objekt bei seiner Erzeugung initialisiert, und einen Destruktor, der aufräumt, sobald die Existenz des Objekts endet, und schon weiß der Compiler, was er zu tun hat. Das spart nicht nur

viel Arbeit, sondern sichert auch die Konsistenz. Man kann darauf vertrauen, daß eine „struct Liste" stets sinnvolle Dateneinträge hat. Dadurch beginnen Datenobjekte tatsächlich, ein gewisses Eigenleben zu führen und so etwas wie eine „ lokale Intelligenz" zu besitzen.

Das populärste Schlagwort der OOP-Fans ist sicherlich die „Vererbung". Das klingt furchtbar sophisticated, ist aber letzten Endes doch sehr trivial.

Um wieder auf das kanonische Beispiel zurückzukommen: Ganz unabhängig davon, ob man in einer Listenstruktur nun Telefonnummern oder spanische Inquisitoren abspeichert, gibt es Probleme und Lösungen, die immer wieder vorkommen. Wenn man ein Datenelement in die Liste einfügt, hängt es durchaus von diesen Daten ab, WO das neue Element eingehängt werden soll, aber das WIE, nämlich dieses furchtbar komplizierte Umhängen von Pointern, ist immer dasselbe. Auch die Löschfunktion könnte man für alle möglichen Listen verwenden.

Der objektorientierte Ansatz ermöglicht es, zunächst ganz allgemein eine Klasse „Listeneintrag" zu definieren und dann davon eine neue Klasse, etwa „Telefonbucheintrag", abzuleiten. Die abgeleitete Klasse „erbt" dann alle Eigenschaften der „Basisklasse" und kann neue hinzufügen, nötigenfalls auch Eigenschaften der Basisklasse in kontrollierter Weise verändern (Stichwort: „virtuelle Member"). Und das ist auch fast schon alles, was es mit der objektorientierten Programmierung, jedenfalls der Art von OOP, die C++ zu bieten hat, auf sich hat, wobei übrigens die oben gewählte Reihenfolge der Konzepte nicht unbedingt mit deren Wichtigkeit zu tun hat.

Ihnen brummt der Schädel? Dann wird es höchste Zeit, daß es konkret wird, und nicht vergessen: Keine Panik!

3.1.2 Member-Funktionen

3.1.2.1 Erste Beispiele

Strukturen sind relativ unhandliche Gebilde, die zudem von Haus aus ziemlich wenig können. Zum Beispiel kann man sie nicht so ohne weiteres über „cout" ausgeben, und so kommt es, daß man oft eine Ausgabefunktion zu einer Struktur schreiben muß. Dies soll als erstes Beispiel für eine Member-Funktion dienen:

```
struct Person
{ char Vorname[20], Nachname[20];
  int Tag, Monat, Jahr;
  void Ausgabe();
};
```

```
Person p1, *person_ptr;
```

Urplötzlich und unerwartet taucht hier in der Strukturdefinition ein Funktionsprototyp auf. Dadurch wird eine Funktion namens „Ausgabe" als Member (oder „Methode") der Klasse „Person" deklariert. Nun kann man sie für jedes Datenobjekt aufrufen:

```
p1.Ausgabe();
person_ptr->Ausgabe();
```

Dafür ist es natürlich wie immer nötig, die Funktion zu definieren. Dabei muß man dem Compiler aber irgendwie klarmachen, daß man die Member-Funktion „Ausgabe" der Struktur „Person" zu definieren gedenkt und nicht irgendeine andere Funktion gleichen Namens, denn genau wie alle anderen Member liegt auch „Ausgabe" hier im Gültigkeitsbereich der Struktur verborgen. Dazu dient der Scope-Operator „::". Mit „Person::Ausgabe" kann man von jeder Stelle des Programms aus genau diese Memberfunktion ansprechen, und das benutzen wir auch, wenn wir sie definieren:

```
void Person::Ausgabe()
{ // usw.
```

Scheinbar hat die Funktion keine Parameter, scheinbar, denn eine normale Memberfunktion hat immer einen Parameter namens „this". Das ist ein Zeiger auf das Objekt, auf dem die Funktion aufgerufen wird. Also können wir folgendes schreiben:

```
void Person::Ausgabe()
{ cout « this->Nachname « ", " « this->Vorname « "\n";
  cout « "geboren am " « this->Tag « "." « this->Monat « "."
  « this->Jahr « "\n";
}
```

Das könnte man schreiben, muß man aber nicht! Die erste Deklaration einer Member-Funktion liegt im Scope ihrer Klasse, und so kommt es, daß auch bei der späteren Funktionsdefinition jener Scope sichtbar ist. Also sind die Membernamen wie „Vorname" oder „Monat" in der Funktion „Person::Ausgabe" direkt sichtbar und bezeichnen dort die entsprechenden Member von „*this". Die folgende Funktion ist viel kürzer und schöner:

```
void Person::Ausgabe()
{ cout « Nachname « ", " « Vorname « "\n";
  cout « "geboren am " « Tag « "." « Monat « "." « Jahr « "\n";
}
```

Auch Member-Funktionen können auf diese einfache Weise aufgerufen werden:

```
struct S
{ int i;
  void f(int);
  int g(int);
};

int S::g(int j)
{ f(j);       // statt "this->f(j);"
  return i;   // statt "this->i;"
}
```

Eine Member-Funktion kann prinzipiell einen Parameter ihres Klassentyps besitzen:

```
struct T1
{ void memberfunktion(T1);
};
```

Beim Aufruf einer solchen Funktion muß aber die ganze Klasse auf den Stack kopiert werden, was bisweilen zu fürchterlich langsamen Programmen führt. Deshalb ist es in der Regel sinnvoller, wenn man als Parameter eine Referenz auf ein konstantes Objekt deklariert:

```
struct T2
{ void memberfunktion(const T2&);  // Meist besser
};
```

Andererseits kann eine Member-Funktion ein Objekt ihres Klassentyps als Ergebnis zurückgeben:

```
struct T3
{ T3 memfunkt();  // OK
};
```

3.1.2.2 Member-Funktionen (Methoden) und konstante Objekte

Da wir gerade beim Thema „const" sind: „this" zeigt ja normalerweise auf ein unqualifiziertes Klassenobjekt, was natürlich Probleme bereitet, wenn man eine solche Funktion auf einem als „const" qualifizierten Objekt aufrufen würde, denn die Memberfunktion könnte das Objekt über „this" beliebig verändern. Deshalb kann man den Typ, auf den „this" zeigt, auf etwas unkonventionelle Weise verändern, indem man nämlich „const" oder „voliatile" an die Funktionsdeklaration anhängt:

```
struct Quali
{ int Member;
  void normal();
  void konstant() const;// d. h. Objekt wird nicht verändert
};

void Quali::normal()
{ Member = 42; }     // OK

void Quali::konstant() const
{ Member = 26731; } // ERROR, "Member" ist hier "const"

void main()
{ Quali q1;
  const Quali q2;

  q1.normal();   // OK
  q1.konstant();// OK

  q2.normal();   // ERROR, "q2" ist "const", "normal" aber nicht!
  q2.konstant();// OK    }
```

Allgemein empfiehlt es sich, mit „const"-Qualifizierungen nicht zu geizen und jede Member-Funktion, die ihr Objekt nicht verändert, als „const" zu deklarieren.

3.1.2.3 Inlining von Member-Funktionen

Member-Funktionen können als „inline" deklariert werden:

```
struct Klasse1
{ int i;
  inline void set_i(int);
  inline int get_i();
};

void Klasse1::set_i(int j)  // „inline" muß man nicht nochmal
{ i = j; }                  // setzen, darf man aber!

int Klasse1::get_i()
{ return i; }
```

Das ist natürlich eine ganze Menge Schreibaufwand für zwei triviale Minifunktionen, weshalb man das auch abkürzen kann: Wenn die Funktionsdefinition direkt in die Klasse hineingeschrieben wird, gilt die Funktion automatisch als „inline":

```
struct Klasse1
{ void set_i(int j) // Automatisch "inline"
  { i = j;
  }
  int get_i()       // Das auch
  { return i;
  }

  int j;
};
```

Solche „inline"-Definitionen werden zuerst nur syntaktisch analysiert und erst am Ende der Klassendefinition auf Semantik getestet und wirklich definiert. Deshalb können solche Funktionen scheinbar auf Namen zugreifen, die noch nicht deklariert sind, wie es z. B. oben mit „i" geschehen ist.

Sie denken, das Beispiel oben sei ziemlich konstruiert und sinnlos? Denkste, so etwas benutzt man tatsächlich des öfteren, dann nämlich, wenn ein Member „privat" ist und man „von außen" aus Sicherheitsgründen nur kontrolliert durch derartige Funktionen darauf zugreifen können soll, was wie immer eine elegante Überleitung zum folgenden Abschnitt darstellt:

3.1.3 Klassen und Zugriffsrechte

Wie zuvor bereits wiederholt angedeutet, unterstützt C++ das Konzept des „Information Hiding" durch unterschiedliche Zugriffsrechte auf Class-Member. Dafür gibt es zunächst die beiden Schlüsselwörter „private" und „public", z. B.

```
struct Namensliste
{ private:
    Namensliste *link;
  public:
    void Einfueg (Namensliste &vorgaenger);
    char Vorname[20], Nachname[20];
};

void Namensliste::Einfueg (Namensliste &vorgaenger)
{ link = vorgaenger.link;
  vorgaenger.link = this;
}

Namensliste nl;

#include <stream.h>
```

```
void main()
{ nl.link = 0;        // ERROR
  cout « nl.Vorname; // OK
}
```

Die Memberfunktion „Einfueg" darf auf alle Member der Struktur zugreifen, einschlielich des privaten Members „link", der so etwas wie einen Verbindungszeiger einer einfach verketteten Liste darstellen soll. Wenn dagegen die Funktion „main", die kein Member der Klasse „Namensliste" ist, an „link" rumfummeln will, kloppt ihr der Compiler auf die Pfoten.

Bei diesem Konzept der Zugriffskontrolle sind zwei Dinge bemerkenswert:

- Es handelt sich wirklich um eine Prüfung auf Rechte und nicht um eine variable Sichtbarkeit. Bei „nl.link" in „main" ist der Bezeichner durchaus sichtbar, und der Compiler weiß, was gemeint ist, so daß MaxonC++ hier durchaus den entsprechenden Code erzeugt. Es wird lediglich gewarnt, daß „main" auf diesen Member nicht zugreifen darf.

- Die Zugriffskontrolle bezieht sich grundsätzlich auf Bezeichner, nicht etwa auf Objekte:

```
struct Beispiel
{ private:
    int i;
  public:
    void rumfummeln(Beispiel*);
};

Beispiel b1, b2;

void Beispiel::rumfummeln(Beispiel *bp)
{ bp->i = 26731;
}

void main()
{ b1.rumfummeln(&b2);
}
```

Im Beispiel wird die Member-Funktion „rumfummeln" auf „b1" mit einem Zeiger auf „b2" als Argument aufgerufen, so daß „this" auf „b1", der Parameter „bp" aber auf ein ganz anderes Objekt zeigt. Trotzdem darf „rumfummeln" an dem privaten Member „i" dieses völlig anderen Objekts 'rumfummeln, und ebenso dürfen alle Objekte einer Klasse unbeschränkt auf alle anderen Objekten derselben Klasse zugreifen.

Es ist natürlich egal, in welcher Reihenfolge man die privaten und die öffentlichen Member einer Klasse deklariert. Nach dem Schlüsselwort „struct" ist der Zugriffsmodus als Default „public" - das muß ja auch so sein, denn sonst wäre C++ ja nicht mehr zu C kompatibel. Aber es gibt daneben auch noch das Schlüsselwort „class", das (fast) identisch mit „struct" ist, mit einem Unterschied: bei „Class" ist „private" der voreingestellte Zugriffsmodus, z. B.

```
class Beispiel
{   int i;    // Ist jetzt automatisch "private"
  public:
    void rumfummeln(Beispiel*);
};
```

Ansonsten ist eine „class" absolut identisch mit einer „struct". Im folgenden werde ich hauptsächlich „class" verwenden, ganz einfach deshalb, weil das so schön objektorientiert klingt und dadurch total hip ist.

Bei überladenen Funktionsnamen dürfen die einzelnen Funktionen durchaus unterschiedliche Zugriffsrechte besitzen:

```
class C
{   void f(int);
  public:
    void f(double);
} c1;

void main()
{ c1.f(1);      // ERROR
  c1.f(1.0);    // OK
}
```

Hier sieht man noch einmal, daß die Zugriffskontrolle in C++ nicht über Sichtbarkeit, sondern über eine tatsächliche Rechte-Prüfung geschieht. Die Funktion „f(int)" ist in „main" sichtbar, so daß der Compiler beim ersten Funktionsaufruf auch erkennt, daß diese Funktion gemeint ist. Wäre sie nicht sichtbar, würde er das Argument „1" einfach nach „double" wandeln und die gerade einzig sichtbare, öffentliche Funktion aufrufen.

Übrigens gibt es noch einen dritten Zugriffsmodus namens „protected", der aber nur im Zusammenhang mit Vererbung nützlich und sonst mit „private" identisch ist. „protected" wird in 3.2.2 näher beschrieben.

3.1.4 No Scope, no hope!

Es ist leider wieder einmal an der Zeit, etwas zur Gültigkeit von Namen zu sagen. Wie Sie sicher schon mitbekommen haben, hat jede Klasse ihren eigenen Scope (Gültigkeitsbereich). In C++ ist das allerdings ein „vollständiger" Scope, während in C nur die Bezeichner der Member darin lagen. Dieser Sachverhalt hat einige unangenehme Konsequenzen, z. B. das kanonische Beispiel, das schon an anderer Stelle dieses Manuals zitiert wurde:

```
class X
{ class Y *ypern;   // 1914
};

class Y
{ int Schmonz;      // ...
}
```

Laut Standard-C++ sind die Klasse „Y", die in „X" referiert wird, und die globale Klasse „Y" absolut verschieden, aber ich erwähnte bereits, daß MaxonC++ solche Fälle beherrscht. Andererseits hat diese Scope-Regelung auch positive Seiten: Man kann „typedef"-Namen und „enum"-Typen deklarieren, die ausschließlich innerhalb einer Klasse sichtbar sind:

```
class Geheim
{ public:
    typedef char *strptr;
    strptr Vorname, Nachname;
    void f();
};

strptr NochEinName;  // ERROR

void Geheim::f()
{ strptr x1;         // OK
                     // usw...
}
```

Der Bezeichner „strptr" liegt im Scope der Klasse „Geheim" und ist deshalb auf Dateiebene nicht sichtbar, wohl aber in der Definition der Member-Funktion „f". Man kann solche Bezeichner aber von außen sichtbar machen, und zwar mit dem Scope-Operator „::", den wir dazu auch schon bei Definitionen von Member-Funktionen benutzen:

```
class Geheim
{ public:
    typedef char *strptr;
    // usw.
};

Geheim::strptr UndNochEinName;   // So geht's!
```

Natürlich gelten hier wieder die bekannten Zugriffsregeln: Wäre „strptr" als privat deklariert gewesen, würde der Compiler wieder eine entsprechende Warnung ausgeben.

Wie bei allen ineinander verschachtelten Scopes, können auch in einer Klasse Bezeichner höherer Ebenen überdeckt werden. Dann kann man mit „::" ohne vorangestellten Klassennamen gezielt Bezeichner der Dateiebene ansprechen:

```
int i;

class S
{ public:
    int i;
    void f();
};

void S::f()
{ i = 1;      // Entspricht "this->i"
  S::i = 2;   // Das auch
  ::i = 3;    // Dies ist das globale "i"
}
```

Einen Namen mit einem „::" bezeichnet man auch als „qualifiziert", was aber nichts mit den Typ-Qualifikationen „const" und „volatile" zu tun hat.

3.1.5 Friends

Das mit den unterschiedlichen Zugriffsrechten ist zwar schön und gut, kann einem aber auch ganz gewaltig auf die Nerven gehen. Betrachten wir wieder einmal das Telefonlisten-Programm. Hier ist es sicher sinnvoll, wenn die beiden Verbindungszeiger der Klasse „Eintrag" privat sind, denn den „Benutzer" dieser Listen soll es möglichst nicht interessieren, wie die Listenstruktur implementiert ist. Andererseits wird man Operationen wie „Einfuegen" wohl als Member-Funktionen der Klasse „Liste" definieren, und die müssen ja normalerweise auf die Link-Zeiger der Listeneinträge zugreifen. Es fehlt also bisher ein Sprachkonstrukt, mit dem man so etwas deklarieren kann wie „Diese

Bezeichner sind privat, aber die Funktion XXX darf sie trotzdem benutzen", und genau dazu dienen „friend"-Deklarationen:

```
class C
{ int member;    // Privat!
  friend void f();
};

C c1;

void f()
{ c1.member++;    // OK
}

void g()
{ c1.member++;    // Fehler
}
```

In der Klasse „C" wird die Funktion „f" als „Freund" der Klasse deklariert, weshalb sie Zugriff auf alle privaten Member von „C" besitzt. „g" ist kein „friend", also bemäkelt der Compiler hier wie gewohnt einen Zugriffsfehler.

Natürlich kann eine Funktion „friend" beliebig vieler Klassen sein. Sie ist nicht Bestandteil der Klasse (wie eine Member-Funktion) und zählt eigentlich auch nicht zu den „Eigenschaften" der Klasse, sondern hat umgekehrt selbst eine Eigenschaft, nämlich die vollen Rechte an der Klasse, deren Freund sie ist. In ihr ist auch der Scope jener Klasse nicht unmittelbar sichtbar.

Bei überladenen Funktionen bezieht sich die „friend"-Eigenschaft ausschließlich auf die Instanz, die als „friend" deklariert wurde:

```
class S
{ int member;
  friend void f();
  friend void f(int);
} s1;

void f()
{ s1.member = 1;
}       // OK

void f(int i)
{ s1.member = i;
}       // OK
```

```
void f(char c)
{ s1.member = c;
}       // ERROR, kein "friend" von "S"
```

Interessanterweise werden an dieser Stelle die sonst so strikten Scope-Regeln durchbrochen: Die Funktion „f" wird bei der „friend"-Deklaration, also innerhalb der Klassendefinition, zum ersten Mal erwähnt, gehört aber trotzdem in den äußeren Scope.

Wenn man einer Member-Funktion einer anderen Klasse die „friend"-Eigenschaft zugesteht, wird das seltsamerweise nicht so locker wie bei globalen Funktionen gehandhabt:

```
class T;

class S
{ friend void T::f();    // ERROR: "f" ist unbekannt
};

class T
{ void f();
};
```

In diesem Fall muß die „friend"-Funktion tatsächlich schon in ihrem eigenen Scope deklariert worden sein, etwa so:

```
class T
{ void f();
};

class S
{ friend void T::f();    // In dieser Reihenfolge OK
};
```

Das führt zu unangenehmen Situationen, wenn zwei Klassen ihren Funktionen gegenseitig den „friend"-Status gönnen wollen:

```
class T;

class S
{ void sfun();
  friend void T::tfun();  // Error
};
```

```
class T
{ void tfun();
  friend void S::sfun();
};
```

Was tun? Einfaches Vertauschen der Klassendefinitionen hilft uns hier offensichtlich nicht aus der Klemme, sondern nur ein neues Feature: Man darf auch eine ganzen Klasse einschließlich aller ihrer Member-Funktionen als Freund deklarieren:

```
class S
{ void sfun();           // Privat!
  typedef char *strptr; // Privat!
  friend class T;        // Ganze Klasse "T" ist Freund
};

class T
{ S::strptr tmember;     // OK, "T" ist friend von "S"
  void tfun();
};

void T::tfun()
{ S s1;
  s1.sfun();             // OK, Freund von "S"
}
```

Man beachte dabei, daß Freundschaft in C++ im allgemeinen eine recht einseitige Angelegenheit ist: Im Beispiel ist „T" ein Freund von „S", aber nicht umgekehrt.

3.1.6 Statische Member

Alles, was mit einer Klasse zu tun hat und ausschließlich von den Member-Funktionen der Klasse benutzt wird, sollte man auch als Member deklarieren und so im Scope der Klasse verstecken. Dabei kommt es allerdings vor, daß ein Objekt oder eine Funktion nicht sinnvoll an ein Objekt gebunden werden kann, sondern eigentlich eher global ist. Um das Gesülze konkret zu machen: Die Klasse „Person", die einen Namen nebst Adresse enthalten soll, hat auch eine Funktion namens „Drucken", die die Daten auf den Drucker ausgeben soll. Dazu braucht sie aber einen Filedeskriptor, der (aus welchen Gründen auch immer) ausschließlich in dieser Klasse benutzt wird und folglich auch dort hinein gehört. Es wäre aber nicht sinnvoll, in jeden einzelnen Datensatz diesen Deskriptor aufzunehmen, denn das wäre ja Speicherverschwendung.

Außerdem braucht man wenigstens noch eine Funktion, die die Druckerdatei öffnet, was natürlich nicht von einem konkreten Objekt abhängig ist und folglich keinen „this"-Zeiger erfordert.

Normalerweise würde man das so implementieren:

```
#include <stdio.h>          // Für Dateien und so'n Zeugs

class Person
{ public:
    char Name[40], Adresse[50];
    void Drucken();
};

FILE *DruckerDatei;         // Filedeskriptor für Drucker

void DruckerOeffnen()
{ DruckerDatei = fopen("prt:", "w"); // Macht Drucker klar
}

void Person::Drucken()
{ fprintf(DruckerDatei, "%s\n", Name);
  fprintf(DruckerDatei, "%s\n", Adresse);
}
```

Mit dem Schlüsselwort „static" kann man nun sowohl die Variable „DruckerDatei" als auch die Funktion „DruckerOeffnen" in den Scope von „Person" verlegen, ohne daß sie an ein Objekt gebunden wären:

```
#include <stdio.h>          // Für Dateien und so'n Zeugs

class Person
{ public:
    static FILE *DruckerDatei;
    static void DruckerOeffnen();
    char Name[40], Adresse[50];
    void Drucken();
};

FILE* Person::DruckerDatei;

void Person::DruckerOeffnen()
{ DruckerDatei = fopen("prt:", "w"); // Macht Drucker klar
}
```

```
void Person::Drucken()
{ // genau wie oben ...
}
```

Auch wenn die beiden Member mit dem Schlüsselwort „static" deklariert werden, haben sie trotzdem externe Linkage, können also von anderen Modulen importiert und benutzt werden. Wir haben es also mit einer weiteren Bedeutung dieses ohnehin schon überladenen Wortsymbols zu tun. Bei einem Datenmember wird sogar „extern" als Speicherklasse angenommen, d. h. der Compiler geht zunächst davon aus, daß diese Variable von woanders importiert werden muß. Erst durch die erneute Deklaration von „Person::DruckerDatei" außerhalb der Klassendefinition wird oben ein entsprechendes Datenobjekt erzeugt.

Der Zugriff auf einen statischen Member erfolgt innerhalb des Scopes der Klasse (also insbesondere aus ihren Member-Funktionen heraus) wie gewohnt, also einfach durch ihren Namen. Von „außerhalb" (und natürlich auch von innerhalb, wenn man das unbedingt will) hat man die Auswahl: Erstens darf man sie wie normale, nicht-statische Member über ein Objekt der Klasse benutzen:

```
Person P1;

void main()
{ P1.DruckerDatei = 0;
   P1.DruckerOeffnen();
}
```

Zweitens sind statische Member bekanntlich nicht an ein bestimmtes Objekt gebunden, weshalb man sie auch über qualifizierte Bezeichner aufrufen kann:

```
void main()
{ Person::DruckerDatei = 0;
   Person::DruckerOeffnen();
}
```

Es versteht sich von selbst, daß auch hier die üblichen Zugriffskontrollen durchgeführt werden, d. h. „main" darf private statische Member von „Person" nicht benutzen.

3.2 Vererbungslehre

3.2.1 Abgeleitete Klassen

3.2.1.1 Ein erstes Beispiel

Das Konzept der Vererbung ist das zentrale Element der objektorientierten Programmierung, und es ist an der Zeit, es hier endlich vorzustellen.

Eine Klasse von Objekten, wie Sie sie bisher kennengelernt haben, repräsentiert die Eigenschaften eines bestimmten Konzepts, einer Idee, vielleicht auch eines Objekts der realen Welt. Dabei macht man jedoch schnell die Beobachtung, daß unterschiedliche Klassen zum Teil identische Eigenschaften haben, daß es oft so etwas wie eine Klassenhierarchie gibt. Ein vielzitiertes Beispiel: Die Klasse „Angestellter" soll die Beschäftigten einer Firma implementieren, etwa so:

```
class Angestellter
{ private:
    // Irgendwelche implementatorische Details
  public:
    char Vorname[20], Nachname[20];
    unsigned int Gehalt;
    // usw..
};
```

Nun gibt es natürlich unterschiedliche Arten von Angestellten; in unserer Firma interessieren uns vor allem Entwickler, Verkäufer und Manager, wobei die Entwickler sich wiederum in Programmierer und Hardwareentwickler unterteilen (Leute für die Produktion brauchen wir nicht, denn wir lassen sowieso in Taiwan produzieren, und der Pförtner hat normalerweise keine besonders aufregenden Eigenschaften und ist deshalb ein ganz normaler Angestellter, wie ihn die gleichnamige Klasse definiert). Jeder Entwickler sollte an einem Projekt arbeiten, deshalb braucht die Klasse „Entwickler" einen zusätzlichen Eintrag für den Projektnamen, während sie sonst alle anderen Eigenschaften der Klasse „Angestellter" übernehmen („erben") soll. Das geht in C++ ganz einfach:

```
class Entwickler : public Angestellter
{ public:
    char Projekt[25];
};
```

Die Klassendefinition beginnt mit einem Doppelpunkt „:" und dem Namen der Klasse „Angestellter", was ganz einfach besagt, daß die neue Klasse „Entwickler" alle Eigen-

schaften der Klasse „Angestellter" erben soll. Man bezeichnet „Angestellter" als „Basisklasse" von „Entwickler", während umgekehrt „Entwickler" eine „abgeleitete Klasse" ist. In anderen Programmiersprachen nennt man den gleichen Sachverhalt „Subklasse" und „Superklasse", was sich aber als nicht sehr glücklich erwies, denn jeder, aber wirklich jeder verwechselt ständig diese beiden Begriffe.

Das „public" besagt, daß die abgeleitete Klasse sich ihrer Herkunft nicht schämt und deshalb die Basisklasse in ihrem öffentlichen Teil liegen soll. Man könnte hier natürlich auch „private" oder „protected" angeben, oder auch nichts - der Default ist „private".

Hinter der Basisklassen-Deklaration folgt eine ganz normale Strukturdefinition, welche die Eigenschaften beschreibt, die die Klasse „Entwickler" zusätzlich zu ihren ererbten haben soll.

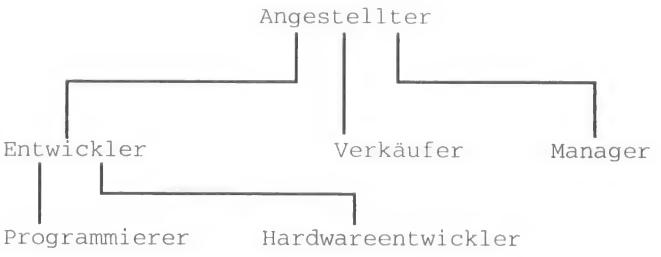
Analog leiten wir die Klassen „Verkaeufer" und „Manager" ab:

```
class Verkaeufer : public Angestellter
{ public:
    int Umsatz;    // Damit wir wissen, wen wir feuern müssen
    int Ort;       // Nummer der Vertriebs-Aussenstelle
};

class Manager : public Angestellter
{ public:
    char KFZ[10]; // Kennzeichen des Dienstwagens
    int Spesenkonto;
};
```

Das Spielchen läßt sich beliebig weiter treiben: Von „Entwickler" leiten wir „Programmierer" ab.

```
class Programmierer : public Entwickler
{ public:
    struct
      { int C:1, Cplusplus:1, Assembler:1, Smalltalk:1 }
      Sprachen;// Bitfeld mit den Sprachen, die er beherrscht
};
```

Die Klasse für die Lötkolben-Aristen erspare ich mir hier, es dürfte wohl auch so klargeworden sein, wie das mit dem Erben und Ableiten geht. Das Resultat ist eine Klassenhierarchie, die man gern mit so komischen Bäumen (komisch deshalb, weil die Wurzel oben ist) veranschaulicht:



Hübsch, nicht? Jedenfalls schulde ich Ihnen noch ein paar Worte darüber, was das Erben von Eigenschaften in der Praxis heißt.

Überall da, wo eine Basisklasse stehen soll, darf auch eine abgeleitete Klasse benutzt werden, z. B.

```
void main()
{ Programmierer P1;

  P1.Sprachen.Cplusplus = 1;

  P1.Gehalt = 5432;     // Eigentlich Member von "Angestellter"

  Entwickler E1 = P1;  // "P1" wird als "Entwickler" betrachtet
                       // und kopiert
}
```

Wie Sie sehen, ist das mit der Vererbung gar nicht so tiefsinnig, wie man manchmal glaubt. Natürlich werden nicht nur die Daten-Member, sondern auch die Member-Funktionen vererbt:

```
class S
{ int i;
  public:
    int get_i();
};

int S::get_i()
{ return i;
}

class T: public S
{ // usw.
};
```

```
void main()
{ T t1;
  int j = t1.get_i();
}
```

Dabei ist es natürlich nicht erforderlich, „T::get_i“ noch einmal zu definieren, denn es wird wieder „S::get_i“ verwendet.

3.2.1.2 Mehrfach-Vererbung und andere Features

Ich erwähnte ja bereits, daß die Symbole „struct“ und „class“ weitgehend identisch sind. Folglich kann man durchaus auch mit Strukturen vererben, allerdings ist bei „struct“ der Zugriffsmodus per Default „public“, so daß man das nicht mehr extra angeben muß. Unionen und Bitfelder können dagegen keine Basisklassen besitzen:

```
struct S
{ int i, j;
};

struct T : S
{ int k:4, l:7;
};          // ERROR

union U : S  // ERROR
{ char c; double d;
};
```

Außerdem muß jede Klasse wirklich definiert sein, bevor man von ihr etwas ableitet:

```
class C;

class D: C     // ERROR
{              // usw.
};
```

Dadurch ist auch ausgeschlossen, daß zwei Klassen voneinander abgeleitet werden, was ja auch wenig sinnvoll ist.

Zu den neuen Features des 2.0-Standards gehört die Mehrfach-Vererbung. Wie der Name schon andeutet, wird dabei eine Klasse von mehreren Basisklassen gleichzeitig abgeleitet, etwa so:

```
class C1
{ int c1;
};

class C2
{ int c2;
};

class C: public C1, private C2
{ int c;
};
```

Jede Klasse kann beliebig viele Basisklassen haben, aber jede nur einmal:

```
class C { };

class D: C, C   // ERROR
{ };
```

Die Zugriffs-Spezifikationen beziehen sich immer nur auf eine einzelne Basisklasse: Bei

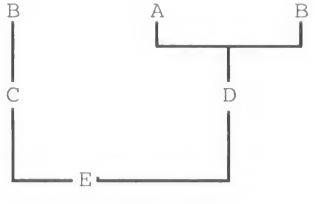
```
class D: public B, C {};
```

hat „C" den Default-Modus, also „private".

Mehrfach-Vererbung funktioniert zunächst auch so, wie man es erwartet, allerdings ist einiges komplizierter als bei Einfach-Vererbung. Man kann jetzt wieder einen Baum malen, der diesmal aber keine Klassenhierarchie, sondern die Basisklassen einer einzelnen Klasse darstellt:

```
struct A { int ia; };
struct B { int ib; };
struct C: B { int id; };
struct D: A, B { };
struct E: C, D { int ie; };
```

ergibt folgenden Graphen für die Klasse „E":



Wie Sie unschwer erkennen, tritt die Klasse „B“ hier doppelt auf, und das ist dann in der Realität auch so. Der Compiler weiß jetzt nicht mehr, was er tun soll, wenn von der Klasse „E“ aus auf den Member „ib“ zugegriffen wird, denn den gibt es ja gleich zweimal:

```
void main()
{ E e1;
  e1.ib = 42;  // ERROR
}
```

Außerdem kann es ja auch vorkommen, daß zwei Basisklassen Member mit identischen Namen vererben:

```
class C1
{ public:
    int x;
};

class C2
{ public:
    char x;
};

class Der: public C1, public C2
{ double usw;
};

void main()
{ Der d;
  d.x = 0;    // ERROR: welches "x"?
}
```

Dem kann man aber durch Qualifizierungen abhelfen:

```
void main()
{ Der d;
  d.C1::x = 0;
}
```

Wie Sie sehen, bereitet Mehrfach-Vererbung bisweilen seltsame Probleme.

3.2.1.3 Vererbung und Typregeln

Wie bereits angedeutet, kann eine abgeleitete Klasse überall dort stehen, wo ein Element einer ihrer Basisklassen erwartet wird. Das trifft auch auf Pointer-Typen zu: Bei dem Beispiel

```
class B
{ int b;
};

class C : public B
{ int c;
};

B b1;
C c1;
```

kann ein Zeiger auf „C" ohne weiteres in einen Zeiger auf „B" umgewandelt werden:

```
B* bp = &c1;
```

Vernünftigerweise wird garantiert, daß „bp" hier auf den Teil von „c1" zeigt, der der Basisklasse entspricht, also wirklich ein „B" ist. In der umgekehrten Richtung geht das im Prinzip auch, allerdings muß man hier casten:

```
C* cp = (C*) &b1;
```

In dieser Form ist das natürlich hanebüchen, denn „b1" ist ja kein Objekt der Klasse „C", und besitzt insbesondere keinen Member „c". Trotzdem ist diese Typkonvertierung erlaubt, denn oft ist sie auch sinnvoll, etwa bei der Rück-Umwandlung von Pointern. Wenn das Programm sich sicher ist, daß der Zeiger „bp" nicht nur auf ein „B", sondern auf den „B"-Teil eines „C" zeigt, kann man ihn wieder umwandeln,

```
C* cp2 = (C*) bp;
```

wobei wieder garantiert wird, daß diese Konvertierung sinnvoll und richtig geschieht, d. h. wenn „bp" wirklich auf den „B"-Teil eines „C"-Objekts zeigt, ist „cp2" anschließend ein Zeiger auf dieses Objekt der Klasse „C".

Natürlich geht das entsprechend auch über mehrere „Generationen", d. h. ein Zeiger auf eine Klasse kann direkt in eine Basisklasse der Basisklasse ihrer Basisklasse verwandelt werden, und mit einem Cast geht es entsprechend umgekehrt. Natürlich wird auch hier bei Mehrfach-Vererbung geprüft, ob die Basisklasse eventuell doppelt vorkommt:

```
struct A { int ai; };
struct B : A {int bi; };
struct C : A {int ci; };
struct D : B, C {int egalwas; };

D d;
A *ap1 = &d;      // ERROR: "A" kommt in "D" mehrfach vor
A *ap2 = (B*) &d; // So geht's
```

Die letzte Initialisierung ist möglich, weil hier der Zeiger auf „d" zuerst in einen Zeiger auf „B" umgewandelt wird, was eine eindeutige Operation darstellt, und erst dann nach „A*" konvertiert, was ebenfalls eindeutig ist. Man muß dem Compiler also manchmal sagen, wie er sich durch einen Klassen-Baum hangeln soll.

Völlig analog gilt das oben Gesagte für Referenzen:

```
class B
{ int b; };

class C : public B
{ int c; };

B b1;
C c1;

B& br = c1;
C& cr = (C&) b1;
```

Schon in 3.2.1.1 wurde darauf hingewiesen, daß man Objekte auch direkt in ihre Basisklassen verwandeln kann, z. B.

```
class B
{ int b; };

class C : public B
{ int c; };

C c1;
B b1 = c1; // Nimm' den B-Teil von c1 und weise ihn b1 zu
```

Es versteht sich, daß bei Mehrfach-Vererbung auch hier geprüft wird, ob die Basisklasse eindeutig ist.

Der umgekehrte Fall,

```
C c2 = (C) b1;  // ERROR
```

ist glücklicherweise nicht erlaubt, denn hier weiß der Compiler, daß „b1" ein „B"-Objekt ist und garantiert nicht zu einem „C"-Objekt gehört.

3.2.2 Scoperegeln und Zugriffskontrolle bei Vererbung

3.2.2.1 Noch mehr Details

Bekanntlich hat jede Klasse ihren eigenen Scope, und das gilt auch bei Vererbung. Eine abgeleitete Klasse kann deshalb problemlos Namen aus ihren Basisklassen überdecken:

```
class Base
{ public: int member;
};

class Derived : Base
{ public: void member(char*);
};
```

Im Zweifelsfall kann man wie gewohnt durch Qualifizierungen klarstellen, was gemeint ist:

```
void main()
{ Derived d;
  char *str = "Niemand erwartet die spanische Inquisition!";

  d.Base::member = 26731;
  d.Derived::member(str); // "d.member" hätte hier auch gereicht
}
```

Trotzdem gibt der Compiler beim obigen Programm eine Fehlermeldung aus, denn er ist der Meinung, daß die Funktion „main" nicht auf den Namen „member" aus „Base" zugreifen darf. Das kommt etwas unerwartet, denn schließlich ist „member" in „Base" doch eigens als „public" deklariert worden. Des Rätsels Lösung ist, daß „Base" eine private Basisklasse von „Derived" ist (Sie erinnern sich: hinter „class" ist der Zugriffsmodus defaultmäßig „private"). Das bedeutet, daß alle Member aus der Basisklasse „Base" in „Derived" privat sind, ja sogar die auf Basisklasse als solche von außen nicht zugegriffen werden kann:

```
Derived d;
Base * bp = &d;   // ERROR: Basisklasse ist privat!
```

Dieses Konzept ist auf den ersten Blick zwar eigenartig, aber doch schlüssig: Wenn der Umstand, daß eine Klasse von einer anderen abgeleitet ist, zum implementatorischen Teil gehört, ist es natürlich sinnvoll, die Basisklasse zu „verstecken" und jeden Zugriff von der Benutzerseite zu unterbinden.

3.2.2.2 Der Zugriffsmodus „protected"

Bisher (und das war auch im „1.0-Standard" so) haben wir nur zwischen den internen Bereichen einer Klasse („private") und der Schnittstelle zur Außenwelt („public") unterschieden. Abgeleitete Klassen gehören in diesem Sinne zur „Außenwelt" ihrer Basisklassen, denn eine abgeleitete Klasse kann natürlich nicht auf die privaten Member ihrer Basisklasse zugreifen. Das ist sinnvoll, aber nicht wirklich befriedigend. Kurz gesagt, es entstand im Laufe der Zeit das Bedürfnis, einen Unterschied zwischen abgeleiteten Klassen und Code, der wirklich außerhalb der Klasse liegt, zu differenzieren. Das Ergebnis dieser Überlegungen war der Zugriffsmodus „protected", der dann in den „2.0-Standard" aufgenommen wurde.

„protected" wird genau wie „public" und „private" benutzt. Auf einen Bezeichner mit diesem Zugriffsmodus können abgeleitete Klassen zugreifen, sonst gilt er als „private". Ein Beispiel:

```
class Klasse
{ protected:
    int imember;
};

class Abgeleitet : public Klasse
{ void set(int i)
  { imember = i;        // OK, da abgeleitete Klasse
  }
};

void main()
{ Klasse k;
  k.imember = 3718847;   // ERROR
}
```

Das ist ein ganz nützliches Feature, wenn man Klassen hat, die mehr oder weniger ausschließlich dafür vorgesehen sind, daß von ihnen weitere Klassen abgeleitet werden.

Da wir jetzt schon drei Zugriffsarten haben, ist es wohl an der Zeit, noch einmal klarzustellen, welche Auswirkungen der Zugriffsmodus einer Basisklasse hat:

Wenn die Basisklasse, wie oben geschehen, als „public" deklariert wird, darf von jeder Programmstelle auf die Basisklasse zugegriffen werden, d. h. die Eigenschaft, daß die Klasse von einer anderen abgeleitet ist, ist kein implementatorisches Detail, sondern öffentlich. Natürlich behalten die Member, die als „private" oder „protected" deklariert sind, diesen Modus auch bei.

Eine Basisklasse, die „private" ist, ist auch nicht weiter schwer zu verstehen: Alle Member bzw. Basisklassen dieser Basisklasse sind absolut privat, sogar die Tatsache, daß abgeleitet wurde, ist für jeden Zugriff von außen tabu, so daß man z. B. keinen Zeiger auf die Klasse in einen Zeiger auf die Basisklasse verwandeln kann - nur die abgeleitete Klasse selbst (bzw. ihre Member-Funktionen und Friends) dürfen das. Und wenn last not least der Zugriffsmodus der Basis „protected" ist, hängt es von Standpunkt des Betrachters ab, ob „private" oder „public" gilt: In der abgeleiteten Klasse wirkt diese Basisklasse wie „public", für Klassen, die wiederum davon abgeleitet werden, bleibt der Zugriffsmodus „protected", und für alles, was nicht zu diesen Klassen oder ihren Friends gehört, ist keinerlei Zugriff auf die Basisklasse erlaubt, so als wäre sie „private". Aber im Grunde genommen hätte ich mir diese ganzen Regeln auch sparen können, denn wenn man einmal scharf darüber nachdenkt, ist die ganze Zugriffsregelung intuitiv völlig klar.

Daran ändert auch das folgende Faeture nichts wesentliches:

3.2.2.3 Zugriffs-Deklarationen

Für manche Sicherheitsfetischisten (Vertrauen ist gut, Kontrolle besser) sind die bisherigen Regelungen für die Zugriffsrechte noch nicht umfangreich genug, und deshalb gibt es auch für sie ein nettes kleines Spielzeug: Zugriffs-Deklarationen für abgeleitete Klassen.

Man stelle sich folgende Situation vor: Sie wollen von einer Klasse ableiten und die Basisklasse als „private" oder „protected" vor unbefugten Zugriffen schützen, aber trotzdem den einen oder anderen Member aus der Basisklasse öffentlich zugänglich machen. Das folgende Programm illustriert, wie das geht:

```
class Bas
{ private:
    int a;
  protected:
    int b;
  public:
    int c, d;
};
```

```
class Der : private Bas
{ public:
    int e;
    Bas::c;
};

void main()
{ Der x;
  x.c = 42;    // Geht, da in "Der" public
  x.d = 4711;  // ERROR, private Basisklasse
}
```

Der Zugriff auf „d" über „Der" ist erwartungsgemäß verboten, denn die Basisklasse „Bas" ist privat. Dagegen kann „c" auch über „Der" benutzt werden, denn er wird dort ausdrücklich als öffentlich deklariert.

Wie Sie sehen, erfolgt eine solche Zugriffs-Deklaration ganz einfach über den Namen des Members ohne jegliche Typangabe. Der neue Zugriffsmodus muß derselbe sein, den der Member auch in der Basisklasse hatte, denn alles andere ist ziemlich sinnlos:

```
class B1
{ private:
    int i;
  public:
    int j;
};

class C1 : private B1
{ public:
    B1::i;    // ERROR
};

class C2: public B1
{ private:
    B1::j;    // ERROR
};
```

Der erste Fehler ist klar, man kann den Schutz eines Members nicht so ohne weiteres verringern, denn das würde ja das Ganze System der Zugriffs-Kontrolle ab absurdum führen, nach dem Motto: Wenn ein Member privat ist, leitet wir einfach eine Klasse ab und setzen uns die Rechte so, wie wir sie haben wollen. Der zweite Fehler ist auf den ersten Blick nicht so offensichtlich, denn was soll schlecht daran sein, einen Member sicherheitshalber privat zu machen? Der Haken ist nur, daß das ganz einfach sinnlos ist: Wenn man einen Objekt der Klasse „C2" hat, nimmt man einfach einen Zeiger darauf, wandelt diesen in einen Zeiger auf „B1" ab (geht, denn schließlich ist „B1" hier öffentliche Basisklasse) und greifen darüber auf „j" zu - alles null Problemo.

Da der Zugriffsmodus also nicht wirklich verändert werden kann, folgt daraus, daß Zugriffs-Deklarationen bei öffentlichen Basisklassen (wie „C2“ im obigen Beispiel) nutzlos sind, denn hier behalten alle Member der Basisklasse ohnehin ihren alten Modus.

Einen eigentümlichen Effekt kann man bei Zugriffs-Deklarationen für überladene Funktionen beobachten: Da die Rechte hier ausschließlich über den Namen gesetzt werden, müssen alle Funktionen dieses Namens denselben Zugriffsmodus haben:

```
class Basis
{ private:
    int g();
    int g(char*);
    void f(int);
  public:
    void f(char);
};

class Klasse : private Basis
{ private:
    Basis::g;     // OK, beide "private"
  public:
    Basis::f;     // ERROR
};
```

Durch die „public“-Deklaration würden beide „f“-Funktionen in „Klasse“ als öffentlich deklariert werden, aber das gibt natürlich Ärger, denn „f(int)“ war ja vorher privat. Auch eine Deklaration als „private“ würde nicht helfen, denn dann würde wiederum der Zugriffsmodus von „f(char)“ verändert. Zugriffs-Deklarationen für überladene Funktionsnamen sind also nur erlaubt, wenn alle Funktionen (wie „g“) den gleichen Zugriffsmodus haben.

3.2.3 Virtuelle Member-Funktionen

3.2.3.1 Ein Beispiel

Lassen wir uns zusammenfassen, was wir bisher an objektorientierten Features kennengelernt haben: Wir können Funktionen an Strukturen anbinden, haben eine Möglichkeit festzulegen, welcher Programmteil auf welchen Member zugreifen darf, und können Strukturtypen durch Vererbung erweitern oder auch zusammenfassen. Das ist alles sehr nützlich, aber irgendwie doch noch nicht so besonders schlau. Aber das kommt jetzt, aber hallo.

Angenommen, Sie haben einen Zeiger auf ein Objekt einer bestimmten Klasse. Dann wissen Sie ja bereits, daß dieser Zeiger in Wirklichkeit durchaus auf ein Objekt einer ganz anderen Klasse zeigen kann. Vielleicht erinnern Sie sich noch an das allererste Beispiel aus Abschnitt 3.2.1.1, an die Abgestellten und Manager und so weiter. Nehmen wir an, die einzelnen „Angestellten" sind in einer Liste angeordnet. Nun haben wir einen Zeiger auf ein solches Objekt und wollen gerne die Daten ausgeben, und zwar vollständig, und dazu müssen wir wissen, von welcher Klasse ein Objekt wirklich ist, also ob wir es mit einem Programmierer, Manager oder bloß einem einfachen Angestellten zu tun haben.

Es wird Sie vielleicht enttäuschen, daß es in C++ keine direkte Möglichkeit gibt, genau dieses festzustellen, denn dann müßte bei wirklich jedem Objekt in irgendeiner Form dessen Typ abgespeichert werden, und das ist erstens gar nicht so einfach und würde zweitens Speicherplatz kosten. Aber das ist eigentlich nicht so schlimm, denn normalerweise muß man nicht wirklich wissen, mit welcher Klasse man es zu tun hat, sondern nur, welche momentan relevanten Eigenschaften diese hat.

So, da Sie jetzt offenbar den vorherigen Satz in seiner ganzen Bedeutungsschwere verarbeitet haben, ist es langsam wieder an der Zeit, konkret zu werden (wer „A" sagt, muß auch „ngst" sagen). Also der Schlüssel dazu sind virtuelle Memberfunktion.

Eine virtuelle Memberfunkion sieht zuerst wie eine ganz normale Funktion aus, nur daß sie mit dem Schlüsselwort „virtual" deklariert wird:

```
#include <stream.h>

class Angestellter
{ public:
    char Vorname[20], Nachname[20];
    unsigned int Gehalt;
    virtual void Ausgabe();
};

void Angestellter::Ausgabe()
{ cout « Vorname « " " « Nachname « "\n";
  cout « "Gehalt: " « Gehalt « " DM\n";
}
```

Ganz normal ist diese Funktion aber keineswegs: In Wirklichkeit trägt der Compiler jetzt bei jedem Objekt der Klasse „Angestellter" einen Vermerk ein, daß immer dann, wenn auf diesem Objekt die Funktion „Ausgabe" aufgerufen wird, die Funktion „Angestellter::Ausgabe" genommen werden soll (vorerst können Sie sich das so vorstellen, als würde in jedes Objekt automatisch ein Zeiger auf diese Funktion eingetragen - die Realität ist ein wenig komplizierter, aber im Prinzip ist das so). Jetzt werden Sie sich fragen, was denn wohl sonst aufgerufen werden soll, wenn nicht

„Angestellter::Ausgabe"? Nun, wir leiten jetzt die Klasse „Entwickler" ab, die eine eigene Ausgabefunktion desselben Namens tragen soll:

```
class Entwickler : public Angestellter
{ public:
    char Projekt[25];
    void Ausgabe();
};

void Entwickler::Ausgabe()
{ Angestellter::Ausgabe();
  cout « "arbeitet an: " « Projekt « "\n";
}
```

Die Klasse „Entwickler" hat ihre eigene Funktion „Ausgabe", die zunächst die „Ausgabe"-Funktion aus der Basisklasse aufruft und dann noch den Namen des Projekts ausgibt. So weit scheint alles noch völlig normal, aber durch den Umstand, daß die Funktion „Ausgabe" denselben Namen trägt wie eine virtuelle Funktion in der Basisklasse, ist sie automatisch ebenfalls virtuell (man hätte hier genau so gut auch ein „virtual" hinzufügen können). Also erhält auch jedes Objekt der Klasse „Entwickler" einen Zeiger (bzw. so etwas ähnliches) auf die Funktion „Entwickler::Ausgabe. Aber jetzt kommt der Trick: In Objekten der Klasse „Entwickler" wird auch der entsprechende Zeiger innerhalb ihrer „Angestellter"-Basisklasse auf diese Funktion gesetzt! Nehmen wir also an, wir haben einen Zeiger auf „Angestellter", z. B. als Parameter einer Funktion:

```
void f(Angestellter *a)
{  a->Ausgabe();
}
```

Beim Aufruf „a->Ausgabe" hat der Compiler natürlich keine Ahnung, ob „a" nun auf einen Angestellten oder einen Entwickler zeigt. Deshalb schaut er nach, auf welche Funktion dort der Zeiger für die Funktion „Ausgabe" zeigt, und springt dahin, also entweder nach „Angestellter::Ausgabe" oder nach „Entwickler::Ausgabe". Ohne virtuelle Funktionen wären die „wahre" Klasse des Objekts „*a" ignoriert und ganz einfach „Angestellter::Ausgabe" ausgeführt worden.

3.2.3.2 Die Details

Es wurde oben behauptet, in C++ könne man nicht direkt feststellen, von welcher Klasse ein Objekt wirklich ist, wenn man bloß einen Zeiger bzw. eine Referenz darauf besitzt. Scheinbar hat der Compiler bei virtuellen Funktionen aber doch diese Möglichkeit. Was ist denn nun?

Bei einer virtuellen Funktion weiß das Programm keineswegs, mit welcher Klasse es zu tun hat, sondern besitzt lediglich Informationen darüber, welche Funktion aufzurufen ist, also, wie ich es oben bereits so schön schwülstig formulierte, welche dazu relevanten Eigenschaften das Objekt hat. Was der Unterschied ist? Nun, eine virtuelle Funktion muß keineswegs auf jeder nachfolgenden Stufe der Klassenhierarchie überdeckt werden:

```
class A
{ public:
    int ai;
    virtual void f();
};

class B: public A
{ int bi; };
```

Hier wird natürlich immer „A::f" aufgerufen, denn „B::f" gibt es ja nicht. Wenn wir weiter ableiten,

```
class C: public B
{ public:
    char ci;
    void f();
};

class D: public C
{  double dd;};

A *ap;
```

gibt es vier Klassen, aber nur zwei Möglichkeiten, was bei „ap->f()" aufgerufen wird: entweder „A::f" (bei Objekten der Klassen „A" und „B") oder „C::f" (Klassen „C" und „D"). Also wird die Wahl der richtigen virtuellen Funktion beim Aufruf nicht anhand der wirklichen Klasse, sondern anhand der höchsten Klasse der Hierarchie, in der die Funktion überdeckt wird, getroffen. Das mag spitzfindig klingen, ist aber der Hauptunterschied in den objektorientierten Konzepten von C++ und Objective C. In Objective C kann man feststellen, von welcher Klasse ein Objekt ist, was in C++ nur mit hohem Aufwand möglich ist, dafür ist C++ wesentlich schneller und braucht weniger Speicherplatz.

Wenn Sie sich die Definition der Funktion „Entwickler::Ausgabe" noch einmal ansehen, werden Sie sich vielleicht fragen, warum es hier nicht zu einer endlosen Rekursion kommt, schließlich wird doch die virtuelle Funktion „Angestellter::Ausgabe" aufgerufen, und die ist bei Entwicklern durch deren Ausgabefunktion überdeckt. Dahinter steckt eine recht sinnvolle Definition: Beim Funktionsaufruf über einen qualifizierten Bezeichner wird diese Funktion grundsätzlich als nicht-virtuell betrachtet, hier also wirklich „Angestellter::Ausgabe" aufgerufen, wie es beabsichtigt war.

Virtuelle Funktionen müssen durchaus nicht immer den gleichen Zugriffsmodus haben:

```
class Base
{ public:
    virtual int f(int);
};

class Derived : public Base
{ private:
    int f(int);
};
```

Sonderlich sinnvoll ist das zwar nicht, aber es ist halt so definiert.

Eine Memberfunktion überdeckt eine virtuelle Funktion einer Basisklasse nur dann, wenn Parameterliste und Ergebnistyp exakt übereinstimmen:

```
class B
{ public:
    virtual int f(char*);
};

class C : public B
{ public:
    unsigned f(unsigned char*);
};
```

Hier ist „C::f" nicht virtuell, denn sie unterscheidet sich in Parameterliste und Ergebnistyp „etwas" von „B::f". Wenn man der Deklaration von „C::f" ein „virtual" vorangestellt hätte, wäre sie zwar virtuell, würde „B::f" aber nicht überdecken.

Eine virtuelle Funktion kann nicht statisch sein, denn beim Aufruf muß anhand eines konkreten Datenobjekts festgestellt werden, welche Funktion zu benutzen ist. Wird sie als „inline" deklariert, muß trotzdem Code für sie erzeugt werden (denn ein Zeiger auf die Funktion muß in die jeweiligen Objekte eingetragen werden) und in der Regel erfolgt der Funktionsaufruf auch nicht über diesen Zeiger und nicht über „inline". Aber MaxonC++ optimiert virtuelle Funktionsaufrufe, bei denen der Objekt-Typ und damit die gewünschte Funktion schon zur Compile-Zeit feststeht, zu gewöhnlichen Funktionsaufrufen:

```
class C
{ int i;
  public:
    virtual void f()
    { i = 42; }
};
```

```
void main()
{ C c1;
  c1.f();          // Nur "C::f" möglich, also kein
                   // Virtueller Aufruf nötig
}
```

Virtuelle Datenmember gibt es übrigens nicht:

```
class Obskur
{ virtual int i;  // ERROR };
```

Falls man so etwas wirklich brauchen sollte, kann man sich mit virtuellen Memberfunktionen, die jeweils eine Referenz auf den passenden Member zurückgeben, behelfen.

3.2.3.3 Abstrakte Klassen

Es gibt Situationen, bei denen man eine Klasse ausschließlich als Basisklasse für andere Klassen benutzen will und nicht die Absicht hat, Objekte dieser Klasse zu erzeugen. Dies kommt insbesondere bei abstrakten Datentypen vor, z. B. kann man sich eine Klasse „Menge" erzeugen und erst durch weiteres Ableiten eine wirkliche Menge mit „echten" Elementen erzeugen. Wenn man aber eine Klasse nie wirklich zum Erzeugen von Objekten, sondern nur zum Vererben benutzen will, wäre es natürlich überflüssig, die virtuellen Funktionen der Basisklasse wirklich zu definieren.

Deshalb hat man im 2.0-Standard „abstrakte Klassen" hinzugefügt. Das Schlüsselkonzept dazu sind die „völlig virtuellen Funktionen".

Nehmen wir an, wir haben unsere Klasse „Mengenelement", die irgendein Element in irgendeiner Menge realisieren soll. Auch hier braucht man, wie so oft, eine Funktion zur Ausgabe dieses Mengenelements, und da wir nichts über den konkreten Mengentypen, den wir später ableiten wollen, wissen, deklarieren wir diese Memberfunktion sinnvollerweise als virtuell.

```
class Mengenelement
{ // Irgendwelche internen Daten, z. B. Listenstruktur
  public:
    virtual void Ausgabe() = 0;
};
```

Das „= 0" hinter der Funktionsdeklaration besagt, daß es sich hier um eine „völlig virtuelle" Funktion handelt, die wir nie zu definieren gedenken, weil es ganz einfach sinnlos ist, ein Mengenelement auszugeben, bei dem wir noch nicht einmal wissen, von welchem Typ die Daten der Menge sind. Dadurch, daß die Klasse eine völlig virtuelle

Funktion besitzt, ist sie für den Compiler eine abstrakte Klasse, er wird nicht zulassen, daß wir ein Objekt dieser abstrakten Klasse erzeugen:

```
void falsch()
{ Mengenelement e1;        // ERROR
  Mengenelement *ep;       // OK
  ep = new Mengenelement; // ERROR
}
```

Aus unserem abstrakten Datentypen erzeugen wir jetzt eine ganz bestimmte Menge, nämlich eine Zahlenmenge, und zu diesem Zwecke brauchen wir den entsprechenden Elementtyp:

```
class ZahlenElement : public Mengenelement
{ public:
    int zahl;        // Das eigentliche Mengenelement
    void Ausgabe();
};

void ZahlenElement::Ausgabe()
{ cout « zahl;
}
```

In der Klasse „ZahlenElement" wird die virtuelle Funktion „Ausgabe" aus der Basisklasse durch eine (implizit zwar virtuelle, aber nicht völlig virtuelle) Funktion überdeckt. Dadurch besitzt die Klasse „ZahlenElement" keine völlig virtuellen Funktionen mehr und ist auch nicht mehr abstrakt. Also können wir Objekte der Klasse „ZahlenElement" nach Herzenslust und ungeniert erzeugen.

Wird von einer abstrakten Klasse eine neue Klasse abgeleitet, ohne alle völlig virtuellen Funktionen der Basisklasse zu überdecken, gilt die neue Klasse ebenfalls als abstrakt. Man kann natürlich davon weiter ableiten und irgendwann auch die letzte völlig virtuelle Funktion richtig deklarieren, um eine „verwertbare" Klasse zu erhalten.

3.2.4 Ein paar Anmerkungen zur Implementation

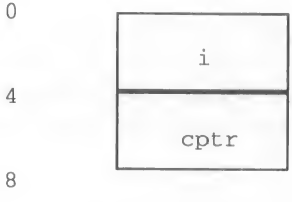
Womöglich sind Sie jetzt völlig verwirrt und fragen sich, was für mystische Dinge ein C++-Programm wohl tut, um alle diese abgefahrenen objektorientierten Features zu implementieren. Sollte das so sein, kann ich Sie beruhigen - all das ist letzten Endes ziemlich simpel. Schließlich ist der objekt-orientierte Teil von C++ nach dem zentralen Prinzip konzipiert, so daß nirgendwo im Vergleich zu „normalen" C-Lösungen ein Overhead an Laufzeit oder Speicherbedarf entstehen soll, eher im Gegentum. Und genau davon soll jetzt die Rede sein.

Fangen wir ganz harmlos an: Eine gewöhnliche Struktur besteht, wie Sie sich denken können, aus einem zusammenhängenden Speicherbereich, in dem die Member liegen, und zwar in der Reihenfolge, in der sie deklariert wurden. Daran ändert sich natürlich nichts, wenn man „class" statt „struct" benutzt. Auch unterschiedliche Zugriffsrechte beeinflussen die Abbildung auf das Zielprogramm keineswegs, denn diese Rechte werden bekanntlich bereits vom Compiler geprüft und folglich besteht kein Bedarf, irgendwelche Angaben darüber in die real existierende Struktur einzutragen. Ach ja, statische Member liegen nicht innerhalb eines Datenobjekts, sondern genau einmal sonstwo im Speicher. Die Deklartion dieser Member innerhalb der Klassendefinition sollte Sie nicht darüber hinwegtäuschen, daß es sich hier um ganz normale statische Daten handelt, nur ihre Sichtbarkeit ist auf eine Klasse beschränkt.

Auch bei Vererbung sieht das keineswegs komplizierter aus. Betrachten wir zunächst Einfach-Vererbung, also Ableitung von nur einer Basisklasse. Ich sagte irgendwo oben, daß man sich Vererbung wie eine Erweiterung einer Struktur vorstellen kann, und genau so realisiert das MaxonC++ auch, und zwar auf die kanonische Weise: Eine abgeleitete Klasse ist immer noch ein zusammenhängender Speicherbereich, wobei der Anfang dieser neuen Struktur der Basisklasse entspricht. Sieht z. B. eine Basisklasse folgendermaßen aus,

```
struct Bas
{ int i;
  char *cptr;
};
```

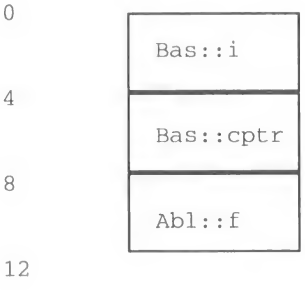
so belegt jedes Objekt dieses Typs acht Bytes, die wie folgt aussehen:



Die Zahlen am linken Rand stellen den Offset innerhalb der Struktur dar, d. h. „i" beginnt an der Startadresse der Struktur und „cptr" vier Bytes weiter. Wenn wir jetzt eine Klasse ableiten, z. B.

```
struct Abl : public Bas
{ float f; };
```

sieht ein Objekt dieser Klasse folgendermaßen aus:

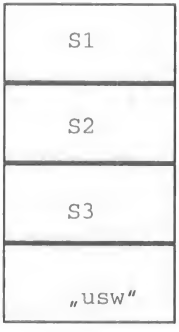


Die Konsequenz ist klar: Wenn man einen Zeiger auf ein solches Objekt hat und ihn als Zeiger auf „Bas" benutzt, muß der Compiler nichts weiter tun, denn die Anfangadresse eines „Abl"-Objekts ist gleichzeitig die Anfangsadresse eines gültigen Objekts der Klasse „Bas", mit allem was dazu gehört.

Bei Mehrfach-Vererbung ist das etwas kompliziereter. Auch hier liegen die Basisklassen am Anfang der abgeleiteten Klasse, aber wie im richtigen Leben kann nur eine die Erste sein. Unter MaxonC++ ist das grundsätzlich die, die zuerst deklariert wurde:

```
struct S1
{ // irgendwas...
};

struct S2
{ // nochwas...
};

struct S3
{ // egal, was...
};

struct T : public S1, public S2, public S3
{ // usw...
}
```

Ein Objekt der Klasse „T" sieht dann so aus (beachten Sie bitte, daß ein Kästchen im folgenden Bildchen nicht wie oben einen einzelnen Member, sondern alle Member einer Klasse darstellt):



Nun hat der Compiler es nicht mehr so leicht: Um einen Zeiger auf „T" in einen Zeiger auf „S2" oder „S3" zu verwandeln, muß er dafür sorgen, daß ein entsprechender Offset addiert wird. Auch bei der umgekehrten Konvertierung (die bekanntlich einen Cast verlangt) muß der entsprechende Offset berücksichtigt, d. h. subtrahiert werden.

Bliebe noch zu erwähnen, wie das mit den virtuellen Funktionen funktioniert. Eine normale, also nicht-virtuelle Member-Funktion ist eine Funktion wie jede andere auch, wobei der „this"-Zeiger als versteckter Parameter übergeben wird. Ein Funktionsaufruf der Form

```
xyz.f(p1, p2, p3);
```

veranlaßt den Compiler lediglich, die Adresse des Objekts zu nehmen und als erstes Argument zu verwenden. Es wird also so etwas erzeugt wie

```
IrgendEineKlasse::f(&xyz, p1, p2, p2);
```

Wenn ein Programm aber erst zur Laufzeit feststellen soll, welche konkrete Member-Funktion aufzurufen ist, braucht sie zweierlei: Einen Zeiger auf die Funktion (Zeiger auf Funktionen gibt es auch außerhalb der OO-Welt) und den richtigen Offset für den „this"-Pointer. Sehen Sie sich noch einmal die obige Abbildung an: Wenn ein Programm einen Zeiger auf ein „S2" hat, das aber in Wirklichkeit nur Bestandteil eines „T" ist, und auf diesem „S2"-Objekt eine virtuelle Member-Funktion, die in „T" überdeckt wird, aufrufen soll, ist der „this"-Zeiger zu modifizieren, denn eine Funktion aus „T" verlangt nun einmal einen Zeiger auf ein „T", was hier etwas anderes als ein Zeiger auf ein „S2" ist. Also muß das Programm nicht nur die Adresse der aufzurufenden Funktion ermitteln, sondern auch noch den richtigen Offset vom „this"-Zeiger subtrahieren.

Jede Klasse kann 739 oder noch mehr verschiedene virtuelle Funktionen haben, und damit wäre es wenig sinnvoll, in jedem Objekt für jede virtuelle Funktion einen Zeiger

und einen Offset anzubringen. Statt dessen legt MaxonC++ für jede benutzte Klassen-Konfiguration eine Tabelle an, in der alle diese Zeiger und Offsets stehen. Ein Zeiger auf diese Tabelle wird automatisch als „unsichtbarer Member" in jedes Objekt eingetragen, sobald es erzeugt wird, z. B. durch eine Variablendeklaration oder mit „new". Später werden wir sehen, daß der Compiler hier einen „Default-Konstruktor" selbsttätig erzeugt und aufruft.

Das ist übrigens auch ein Grund dafür, daß die alte „malloc"-Funktion in C++ durch „new" ersetzt wurde. Der Compiler muß überall dort, wo ein Datenobjekt mit virtuellen Funktionen erzeugt wird, ein paar Zeilen Code erzeugen, die den Zeiger auf die Virtual-Tabelle in das Objekt eintragen. Das kann er aber nur, wenn er weiß, wo welches Objekt erzeugt wird, was mit „malloc" nicht möglich ist, denn das ist nur eine Bibliotheksfunktion wie jede andere auch und der Compiler kann nicht wissen, was sie tut und ob sie ein neues Objekt alloziert oder weiß der Geier was. „new" ist hingegen fester Bestandteil der C++-Syntax und somit ist für den Compiler erkennbar, welches Datenobjekt davon erzeugt wird.

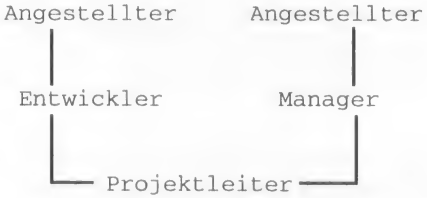
3.2.5 Virtuelle Basisklassen

Oben wurde schon angedeutet, welche - bisweilen immense - Probleme das Konzept der Mehrfachvererbung mit sich bringen kann. Hinzu kommt noch, daß man sie auch nicht so oft braucht oder zumindest mit etwas Mühe durch Einfach-Vererbung ersetzen kann. Ein spezielles Problem wurde aber bisher nur angedeutet: Eine Basisklasse kann innerhalb einer abgeleiteten Klasse in mehreren Instanzen auftreten. Dazu zunächst ein kleines Beispiel, wieder einmal so richtig aus dem Leben gegriffen:

Wir betrachten erneut die Sache mit der Belegschaft unserer ominösen Firma, die zunächst einmal aus lauter Angestellten besteht:

```
class Angestellter
{ private:
    // Irgendwelche implementatorische Details
  public:
    char Vorname[20], Nachname[20];
    unsigned int Gehalt;
    // usw..
};
```

Ferner gab es ja auch noch die davon abgeleiteten Klassen „Entwickler" und „Manager" mit jeweils besonderen Eigenschaften. Zu jedem Projekt gehört aber normalerweise ein Projektleiter, und der soll in unserem Datenmodell die Eigenschaften eines Entwicklers und eines Managers in sich vereinigen. Bei gewöhnlicher Vererbung sieht die Klasse „Projektleiter" dann folgendermaßen aus:



Ei der daus, was ist denn das? Die Basisklase „Angestellter", und mit ihr alle ihre Member, tritt in der Klasse doppelt auf! Das ist natürlich in höchstem Maße unpraktisch, denn erstens werden jetzt die Angestellten-Daten eines Projektleiters doppelt abgespeichert und zweitens weiß der Compiler ohne besondere Qualifizierungen oder Typwandlungen nie so recht, welche der beiden Basisklassen er nehmen soll, wenn die Basisklasse „Angestellter" eines Projektleiters benutzt wird. Das ist schon ziemlich unangenehm.

Solche Probleme löst man durch virtuelle Basisklassen. Wenn eine Klasse eine virtuelle Basis hat, heiß das so viel wie „irgendwo im Datenobjekt will ich so eine Basisklasse haben, aber möglichst nur einmal, und es stört mich nicht, wenn ich mir die Basisklasse mit anderen Klassen teilen muß". Also leiten wir die Klassen „Manager" und „Entwickler" virtuell von „Angestellter" ab:

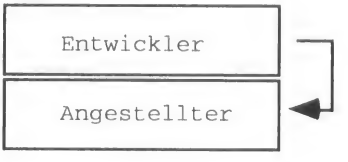
```
class Entwickler : virtual public Angestellter
{ public:
    char Projekt[25];
};

class Manager : virtual public Angestellter
{ public:
    char KFZ[10];   // Kennzeichen des Dienstwagens
    int Spesenkonto;
};
```

Es ist übrigens Banane, ob Sie „virtual public" oder „public virtual" schreiben. In beiden Fällen bezieht sich das „virtual" (genau wie der Zugriffsmodus) ausschließlich auf die nächste folgende Basisklasse, so daß man bei Mehrfach-Vererbung ggf. vor jede Basisklasse ein „virtual" schreiben muß.

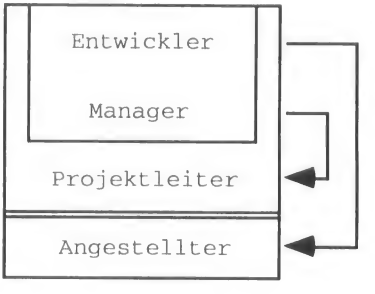
Wenn wir nun den „Projektleiter" von „Entwickler" und „Manager" ableiten, können wir durchaus beide Basisklassen erneut als virtuell deklarieren. Um die ganze Angelegenheit nicht unnötig zu komplizieren, ersparen wir uns das zunächst aber:

```
class Projektleiter: public Entwickler, public Manager
{ public:
    int Bonus;   // Prämie, die er bekommt, falls sein Projekt
          // ausnahmsweise mal termingerecht fertig wird
};
```

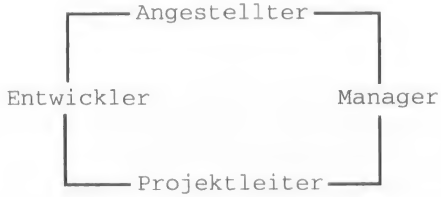
Eine Klasse mit virtueller Basisklasse hat an genau der Stelle, wo eigentlich diese Basisklasse stehen würde, lediglich einen Zeiger auf eine solche Basis. Beim Zugriff auf die virtuelle Basisklasse muß das Programm sich nur an diesem Pointer entlanghangeln. Die virtuelle Basisklasse könnte prinzipiell irgendwo im Speicher stehen, aber sinnvollerweise legt MaxonC++ sie direkt hinter der abgeleiteten Klasse ab, so daß das Objekt einen zusammenhängenden Speicherbereich belegt. Ein Objekt der Klasse „Entwickler" sieht also folgendermaßen aus:



Bei der Klasse „Projektleiter" können wir das Spielchen fortgesetzt: Die beiden Basisklassen „Entwickler" und „Manager" besitzen jeweils einen Zeiger auf ein „Angestellter"-Objekt, das irgendwo am Ende der Klasse steht, aber beide verweisen auf dasselbe Objekt:



Als Ableitungsbaum sieht das so aus:



Anhand dieser über alle Maßen anschaulichen Grafiken wird wohl klar, wie es kommt, daß es in „Projektleiter" nur einen einziges „Angestellter" gibt und jeglicher Zugriff darauf eindeutig ist, auch wenn der Compiler auf dem Weg von „Projektleiter" nach „Angestellter" die freie Auswahl hat, ob er den Zeiger aus „Manager" oder den aus „Entwickler" nehmen will. Das Eintragen dieser Zeiger geschieht übrigens genauso wie das der Zeiger auf die Virtual-Function-Tables, nämlich vollautomatisch, sobald ein neues Objekt erzeugt wird.

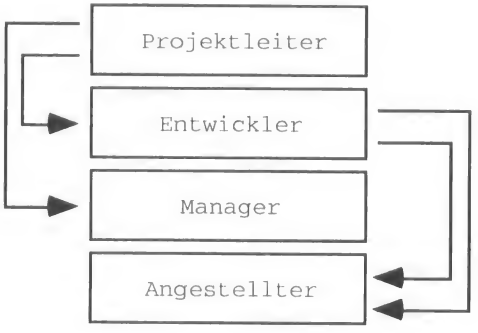
Somit können wir getrost auf alle Member der Klasse „Projektleiter" zugreifen, den Rest macht schon der Compiler:

```
#include <string.h>
#include <stream.h>

void main()
{ Projektleiter P1;

  strcpy(P1.Vorname, "Großer");
  strcpy(P1.Nachname, "Guru");
  P1.Gehalt = 26731;
  strcpy(P1.Projekt, "Grober Unfug");
  strcpy(P1.KFZ, "PB-H 37");
  P1.Spesenkonto = 4711;
  P1.Bonus = 666;

  // mach' irgendwas

  cout « P1.Vorname « " " « P1.Nachname « "\n";
  cout « "Einkommen:    " « P1.Gehalt « " DM\n";
  cout « "Projekt:      " « P1.Projekt « "\n";
  cout « "Dienstwagen: " « P1.KFZ « "\n";
  cout « "Spesen:       " « P1.Spesenkonto « " DM\n";
  cout « "Prämie:      " « P1.Bonus « " DM\n";
}
```

Just for fun überlegen wir uns jetzt noch, wie die Klasse aussehen würde, wenn „Entwickler" und „Manager" ebenfalls virtuelle Basiskklassen von „Projektleiter" wären:

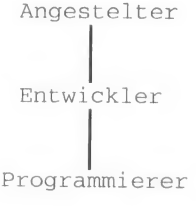
```
class Projektleiter: virtual public Entwickler,
     virtual public Manager ...
```

Nun hat die Klasse „Projektleiter" da, wo vorher ihre beiden Basisklassen hatte, nur zwei Zeiger auf diese Objekte, und diese haben wiederum einen Zeiger auf ihre gemeinsame Basisklasse „Angestellter":



Das Schöne an virtuellen Basisklassen ist ja, daß man sich eigentlich keine Gedanken darüber machen muß, wie das nun funktioniert - man kann ganz einfach darauf vertrauen, daß es klappt und jede virtuelle Basisklasse in jedem Objekt genau einmal existiert. Natürlich haben Sie keine Gewähr, daß der Compiler die Klassen in der oben dargestellten Reihenfolge anordnet. Die virtuellen Basisklassen liegen hinter dem nicht-virtuellen Teil der Klasse, ihre Reihenfolge untereinander ist aber nicht festgelegt.

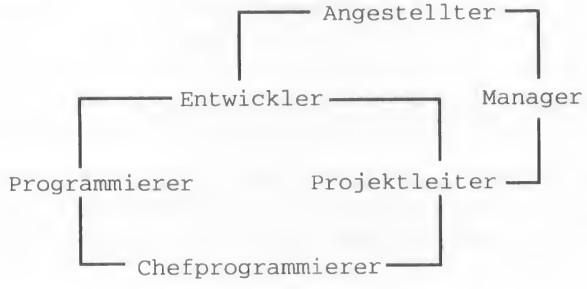
Eine Basisklasse kann innerhalb eines Ableitungsbaums gleichzeitig virtuell und nicht-virtuell sein, z. B. wenn wir „Programmierer" nicht-virtuell von „Entwickler" ableiten und anschließend daraus einen Chef-Programmierer zusammenbasteln:



bzw. im Quelltext:

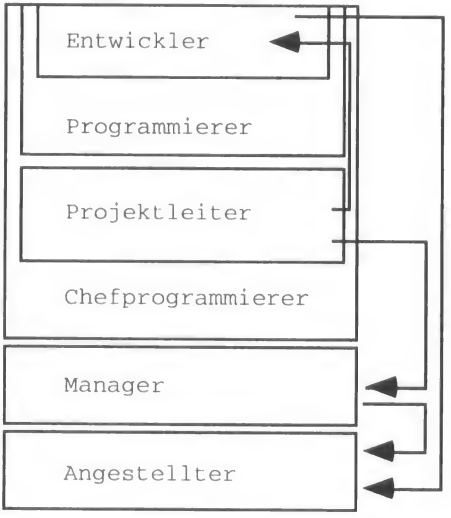
```
class Projektleiter: virtual public Entwickler,
        virtual public Manager ...

class Programmierer: public Entwickler ...

class Chefprogrammierer :public Projektleiter,
         public Programmierer ...
```

Ich habe im Ableitungsbaum absichtlich zwei Fragezeichen gelassen und die Klassendefinitionen nicht endgültig aneinander gesetzt, denn erst muß ich noch erwähnen, daß der Compiler hier merkt, daß die virtuelle Basisklasse „Entwickler" aus der Klasse „Projektleiter" gleichzeitig nicht-virtuell in „Programmierer" vorkommt und deshalb die statische Basisklasse gleichzeitig als virtuelle benutzt:



Na, das ist doch wohl ne Sahne-Klasse, wa? Der Compiler muß jetzt schon ganz schön durch den Klassenbaum navigieren...

Was, Sie wollen ernsthaft wissen, wie das im Speicher aussieht? Na gut, hier ist es:



Wie Sie sehen, handhabt der Compiler Mischformen von virtueller und nicht-virtueller Vererbung durchaus virtuos, aber trotzdem sollte man es vermeiden, damit man nicht selbst den Überblick verliert. Als Faustregel Mehrfach-Vererbung benutzt, grundsätzlich virtuell abzuleiten.

Wenn man nun einen Zeiger auf „Chefprogrammierer" hat und ihn in einen Zeiger auf „Angestellter" umwandeln will, genügt es natürlich nicht, wenn der Compiler einfach einen Offset addiert. Vielmehr muß er sich auf einen von mehreren möglichen Pfaden an Zeigern entlanghangeln, um ans Ziel zu gelangen. Dummerweise führt hier kein Pfeil von „Angestellter" zurück nach „Chefprogrammierer", und durch scharfes Nachdenken wird man einsehen, daß es auch nicht sinnvoll, ja sogar nahezu unmöglich ist, einen solchen Zeiger zusätzlich einzuführen. Deshalb ist es in C++ grundsätzlich nicht möglich, einen Zeiger auf eine Klasse in einen Zeiger auf eine virtuell abgeleitete Klasse umzuwandeln:

```
class VirtBasis
{ char Member; };

class VirtAbgeleitet : virtual public VirtBasis
{ short Noch_n_Member; };
```

```
VirtBasis B1;
VirtAbgeleitet A1;

VirtBasis *bp1 = &A1;       // OK

VirtAbgeleitet *ap1 = (VirtAbgeleitet*) bp1;// ERROR, und
                // zwar trotz Cast!
```

Das klingt vielleicht wie eine gravierende Einschränkung, aber in der Realität wird man so etwas nötigenfalls elegant durch virtuelle Member-Funktionen lösen, denn selbstverständlich kann man die auch bei virtueller Vererbung benutzen.

Apropos virtuelle Funktionen: Bekanntlich hat eine Klasse mit virtuellen Funktionen einen einzigen Zeiger auf eine Tabelle mit den Daten aller virtuellen Funktionen, während für jede virtuelle Basisklasse eine Zeiger darauf direkt in die Klassenstruktur eingesetzt wird. Warum diese beiden unterscheidlichen Lösungen? Könnte man nicht auch die Aufrufsdaten für die Funktionen direkt in die Struktur einsetzen oder umgekehrt auch virtuelle Basisklassen über eine Tabelle verwalten und dann nur noch einen Zeiger auf diese Tabelle in dem Klassenobjekt aufbewahren, vielleicht sogar eine gemeinsame Tabelle für virtuelle Funktionen und virtuelle Basisklassen einführen?

Im Prinzip sind beide Alternativ-Lösungen praktikabel. MaxonC++ verwaltet aber virtuelle Funktionen über Tabellen und virtuelle Basisklassen über direkte Zeiger, und das aufgrund der folgenden praxisnahen Abschätzungen:

1. Laufzeit: Ein Funktionsaufruf ist sowieso eine aufwendige Angelegenheit, und so ist es nicht weiter schlimm, wenn bei einer virtuellen Funktion eine zusätzliche Pointeroperation nötig wird. Dagegen erwartet man als Programmierer, daß der Zugriff auf einen Daten-Member schnell erfolgt, und so ist hier die schnellstmögliche Lösung, nämlich die Zeiger auf die Basisklassen direkt in jedem Klassenobjekt unterzubringen, wohl die sinnvollste.

2. Speicherplatz: Eine Klasse hat bei allen denkbaren Anwendungen nur eine sehr begrenzte Anzahl von Basisklassen, selten wird diese Zahl in der Praxis drei oder vier überschreiten. Da jede dieser virtuellen Basisklassen natürlich selbst einige Daten enthält, kann man davon ausgehen, daß eine Klasse mit z. B. vier Basisklassen einen gewissen Umfang besitzt und die vier zusätzlichen Pointer für die Basisklassen nicht weiter ins Gewicht fallen. Andererseits kommt es vor, daß auf einer Klasse, die nur einige wenige kleine Datenmember enthält, etliche virtuelle Funktionen deklariert sind, und im Laufe des Programms zahllose Objekte dieser kleinen Klasse erzeugt werden. Dann wäre es einfach Wahnsinn, in jedes Objekt für jede einzelne Funktion einen Zeiger und einen Offset einzutragen - der Speicherbedarf wäre enorm.

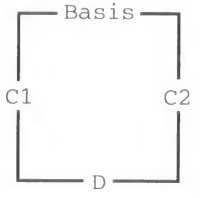
Man beachte, daß die beiden „virtuellen Features" in MaxonC++ (und in jedem anderen C++ wohl auch) so implementiert sind, daß die Grundregel von C++ nicht verletzt wird:

Alles ist so zu implementieren, daß weder in der Laufzeit noch im Speicherbedarf ein Overhead im Vergleich zu nicht-objektorientierten Lösungen entsteht.

Es können schon eigenartige Effekte entstehen, wenn zwei Klassen sich eine Basisklasse „teilen". Dazu gehört auch, daß diese Basisklasse (theoretisch, denn praktisch wird man das wohl vermeiden) unterschiedliche Zugriffsrechte hat:

```
class Basis
{ public:
    int Basismember;
};

class C1 : public virtual Basis { };

class C2 : private virtual Basis { };

class D: public C1, public C2 { };
```

Schauen wir uns ganz schnell den Ableitungsbaum dazu an:



Die Frage, die der aufmerksame Leser sich hier stellen wird, lautet: Ist „Basis" in „D" öffentlich oder privat? Offensichtlich kommt das ganz darauf an, wo man im Baum lang geht, über „C1" oder über „C2". Das kann man in einem Programm durch entsprechende Typwandlungen auch explizit tun:

```
void f(D* dp)
{ dp ->Basismember = 42;          // Nanu?
  ((C1*)dp)->Basismember = 43;  // Über C2, also OK
  ((C2*)dp)->Basismember = 43;  // Über C3, also ERROR
}
```

Wie Sie sehen, ist es möglich, von der Klasse „D" aus auf den Namen „Basismember" zuzugreifen, zumindest mit etwas Aufwand. Deshalb hat man sich entschieden, dort, wo oben „Nanu?" steht, den Zugriff zu erlauben, nach dem Motto, daß es sinnlos ist, Verbote aufzustellen, die man sowieso problemlos umgehen kann.

Ganz allgemein lautet die Regel also: Wenn auf einen Namen auf verschiedenen Wegen mit unterschiedlichen Rechten zugegriffen werden kann, gilt der jeweils geringste Schutz. Ich kann mir aber momentan keine Anwendung vorstellen, bei der man auf die Idee kommen könnte, bei Mehrfach-Vererbung von einer Basisklasse mit unterschiedlichen Zugriffsrechten abzuleiten.

3.3 Konstruktoren und Destruktoren

3.3.1 Initialisierung von Klassen

Lieber Leser, endlich ist Licht am Ende des Tunnels zu sehen: Es fehlt uns nur noch ein wesentliches OO-Konzept von C++, nämlich das der Konstruktoren und Destruktoren.

Als Programmierer steht man eigentlich ständig vor dem Problem, daß ein Datenobjekt initialisiert werden muß, bevor man es benutzen kann. Nun tritt die OO-Programmierung mit dem Anspruch an, daß die Daten des Programms eine gewisse lokale Intelligenz besitzen, und so ist es eine naheliegende Idee, die Initialisierung von Objekten zu automatisieren. In C++ und etlichen anderen Programmiersprachen behandelt man dieses Konzept unter dem Begriff „Konstruktoren".

Ein Konstruktor ist eine Member-Funktion, die automatisch aufgerufen wird, sobald ein Objekt einer Klasse erzeugt wird. Wenn eine Klasse einen solchen Konstruktor besitzt, ist es zwar nicht unmöglich, aber doch ziemlich schwierig, ein nicht initialisiertes Datenobjekt zu erhalten.

Ein Konstruktor kann Parameter haben, und wie jede andere Funktion auch kann man Konstruktoren überladen. Betrachten wir zunächst den einfachsten Fall: Eine Klasse besitzt nur einen Konstruktor, der keine Argumente erwartet. Man kann solche Konstruktoren z. B. benutzen, um alle Member einer Klasse ordentlich mit Null oder einem anderen sinnvollen Wert zu initialsieren. Damit wir aber auch etwas sehen, macht der Konstruktor im folgenden Beispielprogramm auch eine kleine Bildschirmausgabe:

```
#include <stream.h>

class Beispiel
{ public:
    int i;
    char *str;
    Beispiel();  // Das ist ein Konstruktor
};
```

```
Beispiel::Beispiel()
{ i = 0;
  str = "Hello, World!";
  cout « str;
}

void main()
{ Beispiel b1; }
```

Wie Sie sehen, deklariert man eine Konstruktor-Funktion, indem man einer Member-Funktion denselben Namen gibt, den auch die Klasse trägt. Ein Konstruktor darf keinerlei Ergebnistyp haben (nicht einmal „void") und nicht als „static" deklariert werden, was auch einleuchtend ist, denn seine Aufgabe ist schließlich, ein konkretes Objekt zu initialisieren. Aus demselben Grund darf er auch nicht als „const" deklariert werden. Weil ein Konstruktor normalerweise nur dann aufgerufen wird, wenn ein neues Datenobjekt erzeugt wird, weiß der Compiler stets, mit waelcher Klasse er es zu tun hat, und so ist es sowohl sinnlos als auch verboten, einen Konstruktor als virtuell zu deklarieren.

Das Hauptprogramm besteht oben ausschließlich aus einer Deklaration einer Variablen des Typs „Beispiel". Der Compiler weiß aber, daß diese Klasse einen Konstruktor besitzt, und erzeugt gleich bei der Deklaration dieser Variablen einen entsprechenden Konstruktoraufruf, durch den „b1" initialisiert wird. Außerdem hat der Konstruktor im Beispiel noch einen Seitenefekt in Form einer Ausgabe, so daß hier scheinbar aus dem Nichts (auch bekannt als „Nil:") der Text „Hello, World" ausgegeben wird.

Daraus erkennen Sie, daß man Konstruktoren sehr schön dazu mißbrauchen kann, ein Programm durch solche Seiteneffekte vollkommen unlesbar zu machen. Das ist aber natürlich nicht der Sinn der Sache - Konstruktoren sollten ausschließlich zur Initialisierung von Objekten benutzt werden.

Es ist möglich, einen Konstruktor als „private" oder „protected" zu deklarieren. Dann können nur die entsprechend berechtigten Programmteile Objekte dieser Klasse deklarieren.

Zum Schluß sei noch erwähnt, daß die Scope-Regeln hier etwas komisch, aber durchaus sinnvoll deklariert sind, denn die Deklaration eines Konstruktors, der ja immer so heißt wie seine Klasse, verdeckt keineswegs den Klassennamen:

```
class OK
{ OK *zeiger1;
  OK();       // Konstruktor heißt wie Klasse
  OK *zeiger2;  // OK, ist immer noch Klassenname
};
```

Diese Regelung ist zwar inkonsistent, aber sinnvoll.

3.3.2 Konstruktoren mit Argumenten

Ein Objekt mit sinnvollen Default-Werten initialisieren zu lassen, ist zwar ganz praktisch, aber noch nicht die Erfüllung der letzten Träume des Programmierers. Deshalb bietet C++ auch die Möglichkeit, Konstruktoren mit Argumenten aufzurufen und überdies Konstruktoren zu überladen.

Als Beispiel wählen wir uns hier einmal die komplexen Zahlen, die bekanntlich aus einem Realteil und einem Imaginärteil bestehen. Diese beiden Teile sollten stets mit zwei „double"-Werten initialisiert werden:

```
class Complex
{ public:
    double re, im;
    Complex(double, double);
};
```

Jetzt akzeptiert der Compiler es nicht mehr, wenn man ein „Complex"-Objekt „einfach so" deklariert:

```
Complex c1;  // ERROR: No default constructor
```

Statt dessen haben Sie hier gleich zwei verschiedene Möglichkeiten, das Objekt zu initialisieren:

```
Complex c2(1.0, -2.0), c3(0, 0);   // Erste Möglichkeit

Complex c4 = Complex(17.4, 0.42);// Zweite Möglichkeit
```

Die beiden Schreibweisen sind äquivalent, wobei die erste Alternative an Simula angelehnt ist (was, Sie kennen Simula nicht? Na ja, wahrscheinlich waren Sie damals noch zu jung), während die Syntax mit dem „=" mehr Ähnlichkeit mit einer „normalen" Initialisierung in C hat.

Natürlich werden Konsrtuktoren nicht nur bei Variablendeklarationen aufgerufen, sondern nahezu immer, wenn ein Objekt erzeugt wird, und folglich auch bei „new":

```
Complex *cp = new Complex(17.4, 21.0);
Complex *cpvector = new Complex[100];   // ERROR
```

Das zweite „new" ist nicht erlaubt, weil man bekanntlich Vektoren bei „new" nicht initialisieren kann, die Klasse „Complex" aber eine Initialisierung verlangt.

Konstruktoren lassen sich überladen, wie jede andere Funktion auch:

```
class Complex2
{ public:
    double re, im;
    Complex2(double r, double i)
    {  re = r;
       im = i;
    }
    Complex2(double x)
    { re = x;
      im = 0;
    }
    Complex2()
    { re = 0;
      im = 0;
    }
};

Complex2 d1, d2(3), d3(1,2);

Complex2 d4 = 1.2;

Complex2 d5(d1);
```

Hier wurden die Konstruktoren sogar zusätzlich als „inline" definiert, was bei so kleinen Funktionen sinnvoll ist. Natürlich hätte man dasselbe auch durch entsprechende Default Parameter erledigen können, aber Überladen ist nun einmal schöner.

Bei „d4" sehen Sie noch eine dritte mögliche Schreibweise: Wenn ein Konstruktor nur ein Argument erwartet, kann man auf diese Weise auch direkt initialisieren. Auf diese Weise erhält man gewissermaßen eine selbstdefinierte Typkonvertierungsregel.

Bei „d5" wird angedeutet, daß jede Klasse automatisch einen Konstruktor besitzt, nämlich den Kopier-Konstrutor, der ein neues Objekt erzeugt, indem er ein bereits existierendes kopiert. Wenn der Programmierer nicht selbst einen Kopier-Konstruktor der Form

```
class X
{ // ...
  X(const X&);   // Alternativ auch ohne "const"
};
```

deklariert, macht das der Compiler. Das folgende ist kein Kopier-Konstruktor,

```
class Y
{ // ...
  Y(Y);
 };
```

weil für den Aufruf dieser Funktion das Argument auf den Stack kopiert werden müßte, und dazu wäre erneut der Aufruf eines Kopier-Konstuktors notwendig.

3.3.3 Initialisierung von Basisklassen und Membern

Jetzt stellt sich natürlich die Frage, wie ein Konstruktor für abgeleitete Klassen auszusehen hat, schließlich müssen ja auch Basisklassen sauber initialisiert werden. Deshalb hat man für Konstruktor-Definitionen eine besondere Syntax eingeführt. Vor den eigentlichen Anweisungsteil der Funktion sind die Initialisierungen der Basen zu setzen, z. B. so:

```
class A
{ // usw.
  public:
    A(int);
};

class B
{ // usw.
  public:
    B(int, int);
};

class C: public A, private B
{ // Nochwas...
  C(int, int, int);
};

C::C(int a, int b, int c) : A(a), B(b, c)
{ // Sonstige Initialisierungen...
}
```

Im Beispiel hat die Klasse „C" zwei Basisklassen, die beide keinen Default-Konstruktor haben. Deshalb verlangt der Compiler hier, daß „C" einen Konstruktor haben muß, und meldet sonst einen Fehler. Bei der eigentlichen Definition des Konstruktors sind dann die Initialisierungen der Basisklassen entsprechend vorzunehmen. Auch hier wacht der Compiler pingelig darüber, daß keine Basisklasse uninitialisiert bleibt. Wenn eine Basisklasse aber einen Default-Konstruktor hat, kann man ihre Initialisierung weglassen, und jener Konstruktor wird automatisch aufgerufen.

Mit derselben Syntax kann man auch Member initialisieren, was vor allem bei Membern von „const"- oder Referenztypen nötig ist, denn die können nicht durch eine gewöhnliche Wertzuweisung initialisiert werden. Das folgende Listing illustriert dieses Problem und seine Lösung:

```
class Falsch
{ int &ir;
  const int ic;
};        // ERROR: "ir" und "ic" nicht initialisiert

class AuchFalsch
{ int &ir;
  const int ic;
  AuchFalsch(int&);
};              // So weit OK

AuchFalsch::AuchFalsch(int &ref)
{ ir = ref;         // Keine wirkliche Initialisierung!
  ic = ref;         // Verboten, da "const"
}

class Richtig
{ int &ir;
  const int ic;
  Richtig(int&)
};              // OK, hat Konstruktor

Richtig::Richtig(int &ref) : ir(ref), ic(ref)
{ cout « ref;  }
```

Bei solchen Konstruktor-Initialisierungen ist die Reihenfolge von Basisklassen und Membern beliebig. Es versteht sich, daß auch Member mit Konstruktoren auf diese Weise initialisiert werden können - oder sogar müssen, wenn kein Default Konstruktor vorhanden ist:

```
class BuM  // "Basis und Member"
{ int i;
  public:
    BuM(int);
};

class Abgel : public BuM
{ BuM bumm;
  public:
    Abgel(int);  // Klasse MUSS Konstruktor haben!
};
```

```
// Der Konstruktor für "Abgel" konstruiert eine Basisklasse und
// einen Member, beide vom gleichen Typ "BuM":

Abgel::Abgel(int i) : BuM(i+1), bumm(i+2)
{ // Sonstige Initialisierungen ...
}
```

Etwas komisch wird die Angelegenheit bei Mehrfach-Vererbung mit virtuellen Basisklassen:

```
class Basis
{ int i;
  public:
    Basis(int);
};

class Ab1 : public virtual Basis
{ int j;
  public:
    Ab1(int, int);
};

Ab1::Ab1 (int p1, int p2) : Basis(p1)
{ j = p2;   }

class Ab2 : public virtual Basis
{ int k;
  public:
    Ab2(int, int);
};

Ab2::Ab2 (int p1, int p2) : Basis(p1)
{ k = p2; }

class Top: public Ab1, public Ab2
{ public:
    Top(int, int, int);
};

Top::Top (int p, int q1, int q2) : Ab1(p, q1), Ab2 (p, q2)
{ }
```

Die Klasse „Top" hat die Basisklassen „Ab1" und „Ab2", die sich eine gemeinsame Basisklasse „Basis" teilen. Alle Klassen initialisieren über Konstruktoren ihre Basisklassen, was eigentlich zur Folge haben würde, daß „Basis" gleich zweimal initialisiert

würde, was nicht nur Zeitverschwendung wäre, sondern auch zu haarsträubenden Inkonsistenzen führen würde. Stellen Sie sich beispielsweise einmal vor, der Konstruktor von „Basis" würde jeweils eine statische Variable inkrementieren, damit man immer weiß, wie viele Objekte der Klasse „Basis" es gibt. Wenn dann ein Objekt der Klasse „Top", das nur ein „Basis"-Objekt enthält, erzeugt wird, würde dieser Zähler gleich doppelt inkrementiert, was ganz sicher nicht im Sinne des Erfinders ist.

Keine Sorge, genau das passiert nicht. Es wird automatisch geregelt, daß virtuelle Basisklassen nur genau einmal konstruiert werden, und zwar von einer bestimmten abgeleiteten Klasse. Die Regeln, welche Klasse das jeweils ist, sind ziemlich kompliziert und zum Teil implementationsabhängig, deshalb sollten Sie sich darum am besten gar nicht kümmern. Im obigen Beispiel ist es sowieso egal, ob „Basis" von „Ab1" oder „Ab2" aus initialisiert wird, denn in beiden Fällen wird der Konstruktor mit demselben Argument aufgerufen.

Implementiert wird dieses Feature übrigens, indem in Klassen mit einer virtuellen Basisklassen nicht nur der entsprechende Zeiger, sondern auch ein Flag, ob diese Basisklasse zu initialisieren ist, eingetragen wird. Mit demselben Mechanismus wird auch dafür gesorgt, daß eine Klasse genau einmal destruiert wird.

In Konstruktoren und Destruktoren werden Member-Funktionen nicht virtuell aufgerufen:

```
class Base
{ public:
    virtual void f();
};

class Derived : public Base
{ int member;
  public:
    Derived();
};

Derived::Derived()
{ member = 26731;
  f();        // Nicht virtuell!!
}
```

Warum das? Nun, es könnte ja sein, daß von „Derived" weiter abgeleitet wird, etwa so:

```
class D2 : public Derived
{ public:
    int &ir;    // Braucht Initialisierung
    void f();
    D2();
};
```

```
void D2::f()
{ ir = 42; }

D2::D2() : Derived(), ir(member)
{ // usw.   }
```

Basisklassen werden grundsätzlich vor abgeleiteten Klassen konstruiert. Wäre der Aufruf von „f" im Konstruktor von „Derived" virtuell, so würde hier „D2::f" aufgerufen, wo die Referenz „ir" benutzt wird, bevor sie initialisiert wird, und das darf in C++ einfach nicht sein.

3.3.4 Besondere Konstruktoren

3.3.4.1 Default-Konstruktoren

Konstruktoren können beliebige und beliebig viele Argumente haben, ganz wie es für die Initialisierung der Klasse nötig ist. Es gibt aber zwei Arten von Konstruktoren, die aus dem Rahmen fallen: die argumentlosen Default-Konstruktoren und die Kopier-Konstruktoren.

Ein Default-Konstruktor ist ein solcher, der ohne Argumente aufgerufen werden kann. Dafür gibt es mehr Möglichkeiten, als Sie vielleicht denken:

```
class C1
{ public:
   C1();   // Einfachster Fall
};

class C2
{ public:
   C2(...);
};

class C3
{ public:
  C3(int i = 17, double d = 4.0);
};

C1 c1;      // OK
C2 c2;      // OK
C3 c3;      // OK
```

Objekte dieser drei Klassen darf man ohne Argumente bzw. Initialisierung aufrufen, denn der Compiler kann jeweils einen Konstruktor ohne Argumente aufrufen. Als kleine

Inkonsistenz darf man auch Klassen, die überhaupt keinen Konstruktor besitzen, uninitialisiert erzeugen, dann findet entsprechend keinerlei Initialisierung statt.

Vektoren von Klassen lassen sich, sofern man nicht den Default-Konstruktor benutzen will oder kann, auch über Elementlisten konstruieren:

```
class C
{ short nr, x;
  public:
    C(int);   // Kein Default-Konstruktor
};

C cv[5] = { 2, 4, 6, 8, 10 };   // Konstruktor-Aufrufe!
```

Hier wird für jedes Vektorelement ein Konstruktoraufruf mit der entsprechenden Nummer generiert. Diese Syntax kann aber nur eingeschränkt werden, z. B. nicht zur Initialisierung von mit „new" erzeugten Vektoren oder für Vektor-Member:

```
class D
{ C vectormember[3];
  D();      // Konstruktor nötig
}

D::D() : vectormember ( { 0, 8, 15 } ) // ERROR!
{ }
```

Formal wird dieser Fehler damit begründet, daß eine Elementliste im Sinne von C kein „Ausdruck" ist. Wenn Sie sich fragen, wie man obiges Programm richtig schreiben soll, so muß ich leider sagen, daß das unmöglich ist - Member dürfen nur dann Vektoren einer Klasse sein, wenn eben jene Klasse einen Default-Konstruktor besitzt.

3.3.4.2 Kopier-Konstruktoren

Jede Klasse besitzt automatisch einen Kopier-Konstruktor, denn wenn man nicht selbst einen deklariert, tut es der Compiler automatisch.

Normalerweise wird ein Kopier-Konstruktor folgendermaßen deklariert (siehe auch 3.3.2):

```
class C
{ // ...
  C(const C&);
};
```

„Normalerweise“ deshalb, weil es in der Regel nicht nötig sein wird, ein Objekt zu verändern, wenn man es bloß in ein anderes kopieren will. Falls doch, ist

```
class Seltsam
{ // ...
  Seltsam(Seltsam &);
};
```

auch ein gültiger Kopier-Konstruktor. Grundsätzlich sollte man aber Referenzen auf Objekte, die man nicht verändern will, als „const“ deklarieren.

Wenn der Compiler selbst einen Copy-Konstruktor definiert, sieht der so aus, daß alle Member und Basisklassen einzeln kopiert werden. Wenn ein Member oder eine Basisklasse aber einen eigenen Kopier-Konstruktor besitzt, wird natürlich dieser benutzt. Ein vom Compiler generierter Konstruktor ist stets „public“.

Kopier-Konstruktoren werden immer dann aufgerufen, wenn ein Objekt initialisiert wird, nicht jedoch bei einer gewöhnlichen Wertzuweisung:

```
#include <stream.h>

class C
{ public:
    int i;
    C(const C&);
    C(int);
};

C::C(const C &c2)  // Copy-Konstruktor
{ i = c2.i;
  cout « "Copy-Konstruktor ausgeführt.\n";
};

C::C(int i1) : i(i1) { }  // Gewöhnlicher Konstruktor

// Eine Funktion, die ein Objekt als Parameter
// erwartet und zurückgibt:
C f(C carg)
{ C result(carg); // ** 1
  result.i += 42;
  return result;  // ** 2
}
```

```
void main()
{ C c1 = 17, c2 = c1;  // ** 3

  c2 = c1;    // KEIN Copy-Konstruktor-Aufruf

  c2 = f(c1);// ** 4
}
```

In diesem Programm wird der Kopier-Konstruktor genau viermal aufgerufen, nämlich überall da, wo „**" steht:

1. In „f" wird das Objekt „result" deklariert und mit „carg" initialisiert. Unabhängig davon, ob hier die Schreibweise „C x(y)" oder „C x = y" benutzt wird, ist dies der „klassische" Fall für den Aufruf eines Copy-Konstruktors.

2. Bei „return" wird ein Klassenobjekt zurückgegeben. Implementatorisch reicht es natürlich nicht, einen Zeiger auf das lokale Objekt „result" zurückzugeben, denn nach Beendigung der Funktion existiert dieses Objekt ja nicht mehr. Statt dessen muß der Compiler irgendwo ein namenloses temporäres Objekt einrichten, in das das Funktionsergebnis kopiert wird, und weil dieses Phantom-Objekt mit dem Funktionsergebnis initialisiert wird, wird folgerichtig auch bei „return" generell ein Konstruktor aufgerufen.

3. In „main" wird das Objekt „c2" mit dem gleichartigen Objekt „c1" initialisiert, wobei diesmal die Schreibweise mit dem „=" benutzt wird. Auch dies ist ein ganz klarer Fall für den Copy-Konstruktor. In der folgenden Zeile findet dagegen keine Initialisierung, sondern eine ganz gewöhnliche Wertzuweisung statt und deshalb wird dort auch kein Konstruktor aufgerufen. Man kann auch diese normale Wertzuweisung verändern, indem man den Operator „=" überlädt, aber das gehört in ein anderes Kapitel.

4. Zum Schluß wird die Funktion „f" aufgerufen, die als Argument ein Objekt der Klasse „C" erwartet (und nicht etwa eine Referenz darauf). Also wird für diesen Funktionsaufruf ein neues Objekt erzeugt, das für die Funktion später „carg" heißen wird, und natürlich muß dieses neue Objekt konstruiert werden. Also wird „carg" mit dem Copy-Konstruktor aus „c1" erzeugt. Unter (2.) wurde bereits angedeutet, daß das Funktionsergebnis bei „return" in ein temporäres Objekt kopiert bzw. konstruiert wird. Nachher wird dieses temporäre Objekt nach „c2" kopiert, aber das ist wieder eine ganz normale Wertzuweisung und führt deshalb zu keinem weiteren Aufruf des Kopierkonstruktors.

An dieser Stelle möchte ich noch kurz darstellen, wie die Rückgabe eines Klassenobjekts in MaxonC++ intern realisiert wird. Bei numerischen oder Pointerdaten wird das Ergebnis in einem bestimmten Prozessorregister, nämlich „d0", zurückgegeben, aber ein Klassenobjekt wird meist nicht in einem Register Platz haben. Deshalb erhält eine Funktion, die eine Struktur (bzw. ein Objekt) zurückgeben soll, als unsichtbaren Parameter einen

Zeiger auf den Speicherbereich, in den das Ergebnis zu kopieren ist, wobei aber stets der Kopier-Konstruktor verwendet wird. Der Compiler erzeugt bei Bedarf ein temporäres Objekt, in dem das Ergebnis konstruiert wird, und generiert dann Code, der die Daten aus dem temporären Objekt in das eigentliche Zielobjekt kopiert, wobei auch ein überladener „="-Operator Beachtung findet.

Auf den ersten Blick erscheint dieses Hin- und Herkopieren vielleicht ziemlich umständlich, denn man könnte doch auch das Ergebnis direkt im Zielobjekt konstruieren. Aber in C++ wird streng zwischen Initialisierungen und Wertzuweisungen unterschieden, und ein Ausdruck „y = f(x)" ist nun einmal eine Wertzuweisung und keine Initialisierung.

3.3.5 Destruktoren

Konstruktoren stellen ein leistungsfähiges Konzept dar, mit dem man leicht Objekte realisieren kann, die sich automatisch initialisieren und schon bei ihrer Erzeugung komplexe Aufgaben erledigen. Nun bleibt noch das Problem, daß man oft „Aufräumen" muß, sobald ein Objekt zu existieren aufhört. Aber auch das ist in einer Programmiersprache wie C++ kein Problem, denn hier gibt es das Destruktor-Konzept.

Ein Destruktor ist eine Funktion, die der Compiler automatisch immer dann aufruft, wenn ein Objekt der jeweiligen Klasse gelöscht wird, d. h. für statische Variablen am Programmende für Variablen im automatischen Speicher wenn der Block, in dem sie deklariert sind, beendet wird für durch „new" erzeugte Objekte beim Aufruf von „delete".

Man deklariert einen Destruktor als Member-Funktion, die wie ein Konstruktor (also so wie die Klasse) heißt, nur daß man dem Namen ein „~" voranstellt:

```
#include <stream.h>

class Test
{ int nr;
  public:
    Test(int);          // Konstruktor
    ~Test();            // Destruktor
};

Test::Test(int i) : nr(i)
{ cout « "Objekt " « nr « " erzeugt.\n";
}

Test::~Test()
{ cout « "Objekt " « nr « " gelöscht.\n";
}
```

```
void main()
{ Test t1(42), t2(26731);
  cout « "Daten sind initialisiert.\n";
}
```

Das Programm erzeugt folgende Ausgabe:

```
Objekt 42 erzeugt.
Objekt 26731 erzeugt.
Daten sind initialisiert.
Objekt 26731 gelöscht.
Objekt 42 gelöscht.
```

Am Ende der Funktion „main" endet auch die Existenz der Objekte „t1" und „t2", und deshalb erzeugt der Compiler hier automatisch die entsprechenden Destruktoraufrufe - wie man sieht, in der umgekehrten Reihenfolge, in der die Objekte erzeugt wurden. Hier ist ein anderes Beispiel: Ein objekt-orientiertes „Hello World"-Programm:

```
#include <stream.h>

class Hallo
{ public:
    Hallo();
    ~Hallo();
};

Hallo::Hallo()
{ cout « "Hello ";
}

Hallo::~Hallo()
{ cout « "World!";
}

Hallo H;

void main()
{ }
```

Man beachte vor allem, daß hier das alles außerhalb des Hauptprogramms geschieht, nämlich in den Konstruktoren und Destruktoren, die für die globale Variable „H" aufgerufen werden. Solche statischen Variablen mit Filescope werden vor Ausführung von „main" konstruiert und am Programmende destruiert. Im Prinzip kann man auf diese Weise Programme schreiben, die ganz ohne Hauptprogramm auskommen, aber der Linker will trotzdem „main" haben.

Ein Destruktor darf keine Parameter haben - klar, denn der Compiler muß ihn ja selbständig aufrufen und kann sich keine Argumente aus dem Finger saugen. Folglich kann man einen Destruktor nicht überladen, und außerdem darf er kein Ergebnis zurückgeben (noch nicht einmal „void"). Genau wie ein Konstruktor darf ein Destruktor nicht als „static" deklariert werden. Es macht nichts, wenn er „private" oder „protected" ist, der Compiler darf ihn trotzdem von jedem Programmteil aus aufrufen. Bei Destruktoren sagt der Zugriffsmodus nur aus, ob der Programmierer ihn explizit aufrufen darf.

Ein Destruktor wird auch bei „delete" aufgerufen. Im Gegensatz zu „new" ist bei „delete" die genaue Klasse des dynamisch erzeugten Objekts nicht zur Compilezeit bekannt, und deshalb ist es erlaubt und vor allem auch sinnvoll, einen Destruktor als virtuell zu deklarieren:

```
#include <stream.h>

class Basis
{ virtual ~Basis()
  { cout « "Basis destruiert.\n";
  }
};

class Klasse: public Basis
{ virtual ~Klasse()
  { cout « "Klasse destruiert.\n";
  }
};

void main()
{ Basis *bp = new Klasse;

  delete bp;
}
```

Der Destruktor von „Klasse" überdeckt virtuell den von „Basis", und so wird bei „delete bp" der korrekte Destruktor aufgerufen. Die Ausgabe des Programms lautet also:

```
Klasse destruiert.   Basis destruiert.
```

Daraus lernen Sie auch unschwer, daß Basisklassen (und Member) nach Beendigung eines Destruktors automatisch ebenfalls destruiert werden, so daß der Destruktor einer abgeleiteten Klasse sich darum nicht explizit zu kümmern braucht. Destruktoren funktionieren also völlig analog zu Konstruktoren, und deshalb wird es Sie auch nicht überraschen, daß

- Member-Funktionen von Destruktoren aus nicht virtuell aufgerufen werden (siehe 3.3.3),

- beim Löschen von Vektoren von Klassen der Destruktor für jedes einzelne Vektorelement aufgerufen wird, und

- virtuelle Basisklassen bei Mehrfach-Vererbung nur genau einmal destruiert werden (hier werden dieselben Flags wie beim Konstruieren verwendet, siehe 3.3.3).

Ansonsten ist noch zu erwähnen, daß der Compiler Destruktoren (genau wie Konstruktoren) ziemlich clever handhabt:

```
class C
{ public:
    C();
    ~C();
};

void main()
{ goto L1;
  C c1;  // ERROR: Jump past initialisation of "c1"
         // (d. h. es ist nicht mehr klar, ob "c1" später
L1:
         // destruiert werden soll)

  if(1)
  { C c2;
    goto L2;  // Vor "goto" wird "c2" automatisch destruiert
    C c3;   // Destruktor-Aufrufe für "c2" und „c3"
  }

  L2:C c4;
  if (0)
    goto L2;  // Bei "goto": Destruktor-Aufruf für "c4"

  // Hier werden "c4" und "c1" destruiert
}
```

Durch solche Späßchen garantiert der Compiler, daß zu jedem Konstruktor- genau ein Destruktoraufruf erfolgt, ganz so wie man es erwartet.

3.4 Beispielprogramm:

Telefon-Listen, diesmal aber objekt-orientiert

Am Ende von Kapitel 2 habe ich Ihnen als Beispiel ein kleines Telefonbuch-Programm vorgestellt, und am Anfang dieses Kapitels versprach ich Ihnen, daß man so etwas objekt-orientiert viel eleganter lösen kann. Jetzt, da es sowieso wieder einmal Zeit wird, ein Demoprogramm zu bringen, will ich mein Versprechen einlösen: Es folgt eine neue Version des Telefonbuch-Programms und zwar unter Benutzung der meisten objekt-orientierten Features, die bis jetzt vorgestellt wurden.

Ich möchte Sie aber lieber gleich warnen: Das Programm wird dadurch keineswegs einfacher! Ich will hier nämlich zugleich ein Beispiel für abstrakte Datentypen bringen, d. h. zunächst eine ganz allgemeine Listenstruktur realisieren und sie dann durch Vererbung zu einem Telefonbuch spezialisieren. Also, an's Werk:

```
// Objektorientiertes Telefonlisten-Programm

// Dieselben Header-Files wie in der ersten Version:

#include <stream.h>
#include <string.h>
```

Die wichtigste Idee in unserem Programm ist, zuerst zwei Klassen „Listenelement" und „Liste" zu implementieren, die eine doppelt verkettete, sortierte Listenstruktur wie bei der ersten Version realisieren, aber noch nicht festlegen, welche Daten in dieser Liste gespeichert werden sollen. Wir werden davon die beiden Klassen „Telefonbuch" und „Telefonbucheintrag" ableiten und so ziemlich einfach aus der allgemeinen Listenstruktur ein Telefonbuch machen. Ebenso gut könnte man sich aber auch eine nach lateinischen Bezeichnungen sortierte Liste von Würmern implementieren, wobei wiederum jedes Wurm-Objekt als Member eine chronologisch geordnete Liste von Publikationen über ihn enthält, und zu jede Publikation eine Liste der Autoren existiert, oder... Der Phantasie sind keine Grenzen gesetzt, und vielleicht werden Sie auch einmal in einem Ihrer eigenen Programme auf die hier dargestellte abstrakte Listenstruktur zurückgreifen.

Als erstes deklarieren wir die Klasse „Listenelement", die irgendein Element in irgendeiner sortierten, doppelt verketteten Liste darstellen soll. Welche Eigenschaften soll diese Klasse besitzen?

Zunächst einmal wären da die Zeiger „next" und „prev", genau wie im alten Programm. Daran sollte der unbedarfte „Benutzer" der Klasse nicht herumspielen, und deshalb deklarieren wir sie (und einige andere Bezeichenr auch) als „private". Dafür müssen wir,

wie wir später noch sehen werden, die Klasse „Liste" als „friend" deklarieren, denn von dort aus muß auf diese Zeiger zugegriffen werden.

Da die Liste später sortiert sein soll, brauchen wir eine Vergleichsfunktion. Da wir aber noch nichts, aber auch gar nichts über die spätere Liste wissen, deklarieren wir diese Funktion (namens „compare") als virtuell und zwar als voll virtuell, denn zwei ganz allgemeine Listenelemente zu vergleichen, ist sicher nicht sinnvoll. Somit ist die Klasse „Listenelement" auch schon abstrakt, und das ist ja auch so gedacht.

Aus demselben Grund deklarieren wir die Member-Funktion „ausgabe", die die Daten eines Listenelements ausgeben soll, als voll virtuell. Außerdem führen wir der Vollständigkeit halber einen virtuellen Destruktor ein, denn es kann ja sein, daß später eine Klasse mit einem Destruktor von „Listenelement" abgeleitet wird, und so gewährleisten wir in weiser Voraussicht, daß auch stets der richtige Destruktor aufgerufen wird. Wir deklarieremn ihn aber nicht als voll virtuell, sondern definieren ihn als Leerfunktion, so daß man in abgeleiteten Klassen einen Destruktor deklarieren kann, aber nicht muß.

Als kleines Gimmick stellen wir allen abgeleiteten Klassen die Zeiger „next" und „prev" zur Verfügung, und zwar in Form der Funktionen „getnext" und „getprev", die als „protected" deklariert werden. So können abgeleitetet Klassen kontrolliert lesend auf diese Zeiger zugreifen. Wir werden dieses Feature in diesem Programm nicht benutzen.

Wenn wir dann noch bedenkan, daß man im Zweifalsfall alles als „const" deklarieren sollte, um möglichst universell verwendbare Funktionen zu erhalten, sieht die Klassendefinition folgendermaßen aus:

```
class Listenelement
{ private: Listenelement *next, *prev;
  // Vorgänger und Nachfolger

  virtual int compare(const Listenelement*) const = 0;
  // Vergleicht Objekt „*this" mit einem
  // anderen Listenelement
  // und gibt -1, 0 oder +1 zurück (wie bei „strcmp")

  virtual void ausgabe() const = 0;
  // Ein Listenelement ausgeben

  virtual ~Listenelement() { }// Virtuieller Destruktor -
  // hier nur ein Dummy, kann aber
  // in abgeleiteten Klassen überdeckt werden

  protected: // Kleine Extras: abgeleitete Klassen dürfen
             // "next" und "prev" auslesen:
    Listenelement *getnext() const
      { return next; }
```

```
    Listenelement *getprev() const  { return prev; }

    friend class Liste;  // Diese später deklarierte Klasse
// braucht Zugriff auf private Member
};
```

Da hier alle Member-Funktionen entwerder voll virtuell sind (und deshalb nicht definiert werden müssen) oder direkt als „inline" in der Klassendefinition definiert wurden, sind wir mit der Klasse „Listenelement" auch schon fertig und können den abstrakten Datentyp „Liste" entwerfen, der wieder eine ganz allgemeine Listenstruktur darstellen soll und zunächst nichts mit einem Telefonbuch zu tun hat.

Eine „Liste" besitzt wie bisher je einen Zeiger auf das erste und das letzte Listenelement. Auch diese Zeiger sind Privatangelegenheit der Klasse und werden dementsprechend deklariert.

Als nächstes brauchen wir eine Funktion, die ein neues Element an geeigneter Stelle in die Liste einhängt. Die Position kann sie mit der Funktion „compare" aus „Listenelement" herausfinden, aber da wir sonst nichts über den genauen Charakter der Liste wissen, können wir in dieser Struktur kein neues Listenelement erzeugen, sondern brauchen einen Zeiger auf ein fertig alloziertes und initialisiertes Element. Nun gilt es zu verhindern, daß jemand ein von „Listenelement" abgeleitetes Objekt, das einen Fadenwurm repräsentiert, in ein Telefonbuch einfügt - das geht, wenn sowohl die Klasse „Wurn" als auch „Telefonbucheintrag" von „Listenelement" abgeleitet sind. Deshalb deklarieren wir die Funktion „insert" als „protected", d. h. nur die von „Liste" abgeleitete Klasse (z. B. „Telefonbuch") darf neue Elemente in die Liste einfügen.

Eine Liste braucht einen Konstruktor, der die Zeiger „anfang" und „ende" initialisiert (früher hatten wir dafür ja die Funktion „Init", die explizit aufgerufen werden mußte), und einen Destruktor, der alle Elemente der aus Liste löscht. Außerdem deklarieren wir die Member-Funktion „ausgabe", die alle Elemente der Liste (z. B. das gesamte Telefonbuch) ausgeben soll, und Funktionen zum Suchen und Löschen von Listenelementen. Da wir immer noch nicht wissen, mit was für einer Liste wir es später zu tun haben werden, braucht die Suchfunktion einen Zeiger auf ein „Muster"-Listenelement, zu dem sie dann mit der „compare"-Funktion eine Entsprechung in der Liste suchen kann.

Damit kommen wir zu folgender Klassendefinition:

```
class Liste
{ private:
    Listenelement *anfang, *ende; // Erstes und letztes
    // Element Der Liste
  protected:
    void insert(Listenelement *neu);
    // Sucht eine geeignete Position für das
    // Element "*neu" und
    // hängt es in die Liste ein

  public:
    Liste(); // Konstruktor zur Initialisierung
    ~Liste();
    // Destruktor (löscht alle
    // Listenelemente)
    void ausgabe(); // Gibt die ganze Liste aus

  Listenelement *suchen(Listenelement *);
  // Sucht ein Element, das dem
  // Argument-Element "entspricht"
  // (was immer das in einer konkreten Liste heißen mag -
  // jedenfalls ein Element, bei dem "compare"
  // Null liefert)
  // oder Null, wenn kein solches Element existiert.

    void loeschen(Listenelement*);
  // Hängt ein Element aus der Liste aus und löscht es
  // mit "delete"
};
```

Aus didaktischen Gründen implementieren wir jetzt gleich alle Member-Funktionen von „Liste": Zunächst der Konstruktor, der der Funktion „Init" in der alten Programmversion zum Verwechseln ähnlich sieht:

```
Liste::Liste()
{ anfang = 0;
  ende = 0;
}
```

Auch den Destruktor kennen Sie noch, auch wenn er früher eine normale Funktion war und „AllesLoeschen" hieß:

```
Liste::~Liste()
{ Listenelement *p = anfang;

  while(p)
  { Listenelement *hilf = p;
    p=p->next;
    delete hilf;
  }
}
```

Die Member-Funktion „ausgabe" muß hier nichts anderes tun, als für alle Listenelemente deren Funktion „ausgabe" aufzurufen:

```
void Liste::ausgabe()
{ for(Listenelement *l = anfang; l; l = l->next)
    l->ausgabe();
}
```

Dann wäre da noch die Member-Funktion „suchen", die aus einer Liste das erste Element heraussucht, das im Sinne der Funktion „compare" mit dem Vergleichselement übereinstimmt:

```
Listenelement *Liste::suchen(Listenelement *e1)
{ for(Listenelement *e2 = anfang;
    e2 && e2->compare(e1);
    e2 = e2->next);
  return e2;
}
```

Die Funktion „insert" ist zwar etwas komplizierter, aber das meiste kennen Sie ja schon aus dem ersten Telefonbuchprogramm. Im Gegensatz zur alten Funktion „Einfueg" erzeugt sie kein neues Element (wir befinden uns schließlich auf einer Ebene, wo wir noch überhaupt nicht wissen, welche Art von Daten in der späteren Liste abgespeichert werden sollen), sondern hängt nur ein bereits existierendes Element in die Liste ein. Die Position ermittelt sie anhand der Sortierfunktion „compare" aus der Klasse „Listenelement":

```
void Liste::insert(Listenelement *neu)
{ // Position suchen:
  Listenelement *pos = anfang;
  while (pos && neu->compare(pos) > 0)
  pos = pos->next;

  // Element an Position "pos" in Liste einfügen
  // (prinzipiell genau wie in der ersten Programmversion)
```

```
  if (pos == 0)  // Sonderfall: Anfügen an das Ende der
Liste
  { Listenelement *alt = ende; // altes Listenende

    if (alt != 0) // Allgemeiner Unter-Fall: wirklich anhängen
    { alt->next = neu;
      neu->prev = alt;
      neu->next = 0;
      ende = neu;
    }
    else // Spezieller Spezialfall: Liste ist noch leer
    { neu->next = 0;
      neu->prev = 0;
      anfang = neu;
      ende = neu;
    }
  }
  else
  if (pos->prev == 0)  // Sonderfall: Neues Element
    // an Listenanfang hängen
    { pos->prev = neu;
      neu->next = pos;
      neu->prev = 0;
      anfang = neu;
    }
    else // Allgemeiner Fall: Element in Liste einhängen
    { Listenelement *vor = pos->prev;
      // "neu" zwischen "vor" und "pos" einhängen:
      vor->next = neu;
      neu->prev = vor;
      neu->next = pos;
      pos->prev = neu;
    }
  }
```

Und weiter geht's - mit einer Funktion, die ein Element aus einer Liste aushängt und dann mit „delete" vernichtet. Auch diese Funktion leitet sich direkt aus der alten Programmversion ab, und zwar von der Funktion „Loeschen":

```
void Liste::loeschen(Listenelement *E)
  // Löscht das Element, auf das "E" zeigt, aus der Liste
{ // Vorgänger und Nachfolger des zu löschenden Elements
  Listenelement *vor = E->prev, *nach = E->next;
```

```
  if (vor)
    // Element hat Vorgänger: Dann dessen Nachfolger umsetzen
    vor->next = nach;
  else // Kein Vorgänger: Dann Listenanfang aktualisieren
    anfang = nach;

  if (nach)
    // Element hat Nachfolger: Dann dessen Vorgänger verändern
    nach->prev = vor;
  else     // Kein Nachfolger, also Listenende anpassen
    ende = vor;

  delete(E);
}
```

Wow, das war auch schon fast alles - jedenfalls haben wir damit einen leistungsfähigen abstrakten Datentypen implementiert. Jatzt müssen wir nur noch einen ganz konkreten Datentypen, nämlich das bekannte und beliebte Telefonbuch, daraus ableiten.

Zunächst entwickeln wir den Typen „Eintrag" aus „Listenelement", d. h. wir leiten eine neue Klasse ab. Später werden wir sehen, daß es sinnvoll ist, wenn jeder weiß, daß ein „Eintrag" von einem „Listenelement" abgeleitet ist, deshalb erfolgt die Vererbung hier über „public".

Ein Telefonbucheintrag besteht nach wie vor aus einem Namen und einer Nummer, und auch diese sind „public", damit jeder darauf hemmungslos zugreifen kann. Ferner brauchen wir noch einen Konstruktor, denn durch scharfes Nachdenken stellen wir fest, daß es nirgendwo sinnvoll ist, ein uninitialisiertes „Eintrag"-Objekt zu haben. Und last not least müssen wir ja auch noch die beiden voll virtuellen Funktionen „compare" und „ausgabe" definieren, denn „Eintrag" soll ja kein abstrakter Datentyp mehr sein:

```
class Eintrag : public Listenelement
{ public:
    char name[30];    // Name und...
    char nummer[20]; // Telefonnummer

    // Konstruktor:
    Eintrag(const char *neuname, const char *neunummer);

    // Funktionen, die virtuelle Funktionen der Basisklasse
    // definieren:
    int compare(const Listenelement*) const;
    void ausgabe() const;
}
```

Auch hier wollen wir die fraglichen Funktionen gleich definieren. Zunächst einmal wäre da der Konstruktor, der ganz einfach seine beiden Argumente in das „Eintrag"-Objekt kopieren muß:

```
Eintrag::Eintrag(const char *neuname, const char
*neunummer)
:name(neuname), nummer(neunummer) { }
```

Beachtenswerterweise müssen wir hier nicht auf „strcpy" wie im ersten Telefonprogramm zurückgreifen, denn bei dieser Schreibweise haben wir es mit einer echten Initialisierung und nicht mit einer Wertzuweisung zu tun, und folglich ist der Compiler gnädigerweise bereit, die Strings ganz einfach so zu initialisieren.

Die Funktion „compare" soll das Objekt, auf dem es aufgerufen wird, mit dem Element, auf das ihr Argument zeigt, vergleichen und in Abhängigkeit davon -1, 0 oder +1 zurückgeben. Das läßt sich im Prinzip mit einem einfachen „strcmp" der Namensstrings bewerkstelligen, wobei wir nur darüber stolpern, daß eine Funktion eine virtuelle Funktion nur dann überdeckt, wenn die Argumente absolut identisch sind. Also muß die Funktion hier ein „Eintrag" (nämlich „*this") mit einem „Listenelement" (das Objekt, auf das das Argument zeigt) vergleichen. Hier müssen wir ganz einfach darauf vertrauen, daß alle Elemente der Liste zur Klasse „Eintrag" gehören, und uns mit einer simplen Typkonvertierung behelfen:

```
int Eintrag::compare(const Listenelement* that) const
{ Eintrag *that2 = (Eintrag*) that;
  return strcmp(this->name, that2->name);
}
```

Ich hoffe, daß Sie schon jetzt das nötige „Unrechtsbewußtsein" aufbringen, um diese Funktion nicht so recht zu mögen. Wir haben hier einen Zeiger auf ein „Listenelement" und vertrauen darauf, daß es sich in Wirklichkeit um einen „Eintrag" handelt, und zwar nur deshalb, weil wir nicht vorhaben, Objekte anderer Klassen in die Liste einzutragen. Aber andererseits ist es ja auch wenig sinnvoll, einen Telefonbucheintrag mit einem Regenwurm oder einem spanischen Großinquisitor zu vergleichen, und deshalb ist es durchaus sinnvoll, für die Funktion „compare" anzunehmen, daß alle Listenelemente gleicher Art sind.

Außerdem ist es eine verblüffende (und in dieser Form vielleicht nicht für alle Anwendungsgebiete verwendbare) Tatsachen daß wir die Funktion „compare" in unserem Datemmodell für zwei verschiedene Zwecke verwenden: Erstens zum Suchen eines Elemente (Test auf Gleichheit) und zweitens zum Sortieren beim Einfügen (wo es im Wesentlichen auf „kleiner" oder „größer-gleich" ankommt). Schließlich weiß jedes Kind, daß es in Los Angeles mindestens drei verschiedene „Sarah Connor" gibt...

Die folgende Funktion ist wenigstens halbwegs unkompliziert: Sie gibt ein Objekt der Klasse „Eintrag" aus:

```
void Eintrag::ausgabe() const
{ cout « name « ": " « nummer « "\n";
}
```

Jetzt fehlt uns nur noch die Spezialisierung der Klasse „Liste" zum „Telefonbuch". Da wir es bei den Eigenschaften der Klasse „Liste" hauptsächlich mit Low-Level-Features zu tun haben, besteht kein Bedarf, diese Basisklasse öffentlich zu machen. Statt dessen vererben wir hier mit Modus „private" und machen nur die Member-Funktionen „ausgabe" und „loeschen" durch Zugriffsdeklarationen „public".

Auf die Member-Funktion „Liste::insert" sollte der Benutzer der Klasse nicht zugreifen können, weil er dann ja theoretisch einen Fadenwurm in ein Telefonbuch einfügen könnte, wie Sie sich vielleicht erinnern. Also bleibt „Liste::insert" in der privaten Basisklasse verborgen, und wir bieten dem Klassenbenutzer eine Member-Funktion „Telefonbuch::insert" an, die als Argumente zwei Strings erwartet und einen entsprechenden Telefonbucheintrag erzeugt sowie in die Liste einfügt. Auch die Funktion „suchen" definieren wir neu, um erstens ganz einfach nach einem Namen suchen zu können (und nicht anhand eines Vergleichselements) und zweitens die Typgleichheit zu garantieren. Daß wir mit „insert" und „suchen" die entsprechenden Funktionen der Basisklasse namensgleich überdefinieren, hat übrigens keine besondere Bedeutung, denn die gleichnamigen Funktionen der Basisklasse sind ja sowieso „protected" und deshalb von außen nicht erreichbar.

Hier ist also die letzte Klassendefinition unseres Programms:

```
class Telefonbuch : private Liste
{ public:
    Eintrag *insert(const char *, const char *);
    Eintrag *suchen(const char *);
    Liste::ausgabe;
    Liste::loeschen;
};
```

Die beiden Member-Funktionen, die es jetzt noch zu definieren gilt, sind vergleichsweise dröge, da sie hauptsächlich auf Funktionen der Basisklasse „Liste" zurückgreifen - und genau das war ja auch unsere Absicht, denn wenn wir einmal den abstrakten Datentypen „Liste" implementiert haben, wollen wir schließlich ohne größeren Aufwand spezielle Listentypen ableiten können. Zuerst also die neue „insert"-Funktion, die nur noch mit „new" ein neues Element erzeugt, es in die Liste einfügt und einen Zeiger darauf (bzw. Null bei Fehler) zurückgibt:

```
Eintrag* Telefonbuch::insert(const char *neuname, const
char *neunummer)
{ // Neues Element erzeugen:
  Eintrag *neu = new Eintrag (neuname, neunummer);

  if (!neu)
    return 0; // Kein Speicher frei!

  // Element in Liste einhängen:
  Liste::insert(neu);
  // Fertig, Zeiger auf neues Element zurückgeben:
  return neu;
}
```

Die Suchfunktion ist noch viel einfacher: Hier erzeugt man aus dem Suchstring ein Vergleichsobjekt (als automatische Variable!) und sucht dieses Element mit der vorhandenen Funktion in der Liste. Nun muß das Ergebnis nur noch mit einem Cast umgewandelt werden, da „Liste::suchen" einen Zeiger auf ein „Listenelement" zurückgibt, hier aber sinnvollerweise ein Zeiger auf „Eintrag" geliefert werden soll, und schon sind wir fertig:

```
Eintrag *Telefonbuch::suchen(const char *name)
{ Eintrag dummy(name," ");
  return (Eintrag*)
  Liste::suchen(&dummy);
};
```

Ob Sie es glauben oder nicht - das war's! Jetzt folgt noch einmal das ganze Programm im Überblick, einschließlich eines kleinen Hauptprogramms, daß die Benutzung unserer Klassen und Funktionen demonstriert.

```
//  Objektorientiertes Telefonlisten-Programm //
//  mit generischen Listenklassen //


#include <stream.h>
#include <string.h>

// * * * abstrakter Datentyp "Listenelement" * * *

class Listenelement
{ private:
    Listenelement *next, *prev; // Vorgänger und Nachfolger

  virtual int compare(const Listenelement*) const = 0;
```

```
  // Vergleicht Objekt „*this“ mit einem
  // anderen Listenelement
  // und gibt -1, 0 oder +1 zurück (wie bei "strcmp")

  virtual void ausgabe() const = 0;
  // Ein Listenelement ausgeben

  virtual ~Listenelement() { }
  // Virtuieller Destruktor - hier nur ein Dummy, kann aber
  // in abgeleiteten Klassen überdeckt werden

  protected:
  // Kleine Extras: abgeleitete Klassen dürfen
  // "next" und "prev" auslesen:
  Listenelement *getnext() const
  { return next;
  }
  Listenelement *getprev() const
  { return prev;
  }

  friend class Liste;  // Diese später deklarierte Klasse
     // braucht Zugriff auf private Member
};

  // * * * generischer Datentyp "Liste" * * *

class Liste
{ private:
    Listenelement *anfang, *ende;
    // Erstes und letztes Element der Liste
  protected:
    void insert(Listenelement *neu);
    // Sucht eine geeignete Position für das
    // Element "*neu" und
    // hängt es in die Liste ein

  public:
    Liste();  // Konstruktor zur Initialisierung
    ~Liste(); // Destruktor (löscht alle
              // Listenelemente)
    void ausgabe(); // gibt die ganze Liste aus

    Listenelement *suchen(Listenelement *);
    // Sucht ein Element, das dem Argument-Element
```

```
  // "entspricht" (was immer das in einer konkreten Liste
  //  heißen mag - jedenfalls ein Element, bei dem
  // "compare" Null liefert) oder Null, wenn kein solches
  //  Element existiert.

  void loeschen(Listenelement*);
  // Hängt ein Element aus der Liste aus und löscht es
  // mit "delete"
  };

  Liste::Liste()
  { anfang = 0;
    ende = 0;
}

Liste::~Liste()
{ Listenelement *p = anfang;

  while(p)
  { Listenelement *hilf = p;      p=p->next;
  delete hilf;
  }
}

void Liste::ausgabe()
{ for(Listenelement *l = anfang; l; l = l->next)
    l->ausgabe();
}

Listenelement *Liste::suchen(Listenelement *e1)
{ for(Listenelement *e2 = anfang;
  e2 && e2->compare(e1);
  e2 = e2->next);
  return e2;
}

void Liste::insert(Listenelement *neu)
{ // Position suchen:
  Listenelement *pos = anfang;
  while (pos && neu->compare(pos) > 0)
    pos = pos->next;

  // Element an Position "pos" in Liste einfügen
  // (prinzipiell genau wie in der ersten Programmversion)
```

```
    if (pos == 0) // Sonderfall: Anfügen an das Ende der Liste
    { Listenelement *alt = ende; // Altes Listenende

      if (alt != 0) // Allgemeiner Unter-Fall: Wirklich anhängen
      { alt->next = neu;
        neu->prev = alt;
        neu->next = 0;
        ende = neu;
      }
      else // Spezieller Spezialfall: Liste ist noch leer
      { neu->next = 0;
        neu->prev = 0;
        anfang = neu;
        ende = neu;
      }
    }
    else
      if (pos->prev == 0)
      // Sonderfall: Neues Element an Listenanfang hängen
      { pos->prev = neu;
        neu->next = pos;
        neu->prev = 0;
        anfang = neu;
      }
      else // Allgemeiner Fall: Element in Liste einhängen
      { Listenelement *vor = pos->prev;
        // "neu" zwischen "vor" und "pos" einhängen:
        vor->next = neu;
        neu->prev = vor;
        neu->next = pos;
        pos->prev = neu;
      }
    }

void Liste::loeschen(Listenelement *E)
// Löscht das Element, auf das "E" zeigt, aus der Liste
{  // Vorgänger und Nachfolger des zu löschenden Elements
  Listenelement *vor = E->prev, *nach = E->next;

  if (vor)
    // Element hat Vorgänger: Dann dessen Nachfolger umsetzen
    vor->next = nach;
  else
    // Kein Vorgänger: Dann Listenanfang aktualisieren
    anfang = nach;
```

```
  if (nach)
  // Element hat Nachfolger: Dann dessen Vorgänger verändern
    nach->prev = vor;
  else
  // kein Nachfolger, also Listenende anpassen
    ende = vor;

  delete(E);
}

// * * * spezieller Datentyp "Eintrag" * * *

class Eintrag : public Listenelement
{ public:
    char name[30];// Name und...
    char nummer[20]; // Telefonnummer

        // Konstruktor:
    Eintrag(const char *neuname, const char *neunummer);

  // Funktionen, die virtuelle Funktionen der Basisklasse
  // definieren:
    int compare(const Listenelement*) const;
    void ausgabe() const;
}

Eintrag::Eintrag(const char *neuname, const char
*neunummer) : name(neuname),      nummer(neunummer) { }

int Eintrag::compare(const Listenelement* that) const
{ Eintrag *that2 = (Eintrag*) that;
  return strcmp(this->name, that2->name);
}

void Eintrag::ausgabe() const
{ cout « name « ": " « nummer « "\n";
}

// * * * spezieller Datentyp "Telefonbuch" * * *

class Telefonbuch : private Liste
{ public: // Neues Element erzeugen und einfügen:
    Eintrag *insert(const char *, const char *);
```

```
  // Element anhand des Namens suchen:
    Eintrag *suchen(const char *);

  // Zwei Funktionen werden öffentlich vererbt:
    Liste::ausgabe;
    Liste::loeschen;
};

Eintrag* Telefonbuch::insert(const char *neuname, const
char *neunummer)
{ // Neues Element erzeugen:
  Eintrag *neu = new Eintrag (neuname, neunummer);

  if (!neu)
    return 0; // Kein Speicher frei!

  // Element in Liste einhängen:
  Liste::insert(neu);

  // Fertig, Zeiger auf neues Element zurückgeben:
  return neu;
}

Eintrag *Telefonbuch::suchen(const char *name)
{ Eintrag dummy(name,"");
  return (Eintrag*)
  Liste::suchen(&dummy);
};

// * * *  Hauptprogram  * * *

void main()
{ // Eine Liste wird deklariert und initialisiert:

  Telefonbuch t;

  // Ein paar Beispieldaten werden eingefügt:

  t.insert("Harley-Davidson", "001-414/342-4680");
  t.insert("Boris Becker", "0815/4711");
  t.insert("James Bond", "007/26731");
  t.insert("Maxon", "06196/481811");
  t.insert("Zaphod Beeblebrox", "00042/08154242");
  t.insert("Luxemburg", "00352");
  t.insert("Queensr\0377che", "795069");
  t.insert("Fishbone", "4676152");
```

```
    // Ein Datensatz wird gesucht und ggf. gelöscht:

  Eintrag *e = t.suchen("Maxon");
  if (e)
    t.loeschen(e);
  else
    cout « "Name nicht gefunden!\n";

  // Restliche Liste ausgeben:
  t.ausgabe();

  // Löschen der Liste geschieht automatisch durch Destruktor-Aufruf
}
```

3.5 Noch mehr Features

3.5.1 Zeiger auf Member

Zeiger auf Member stellen eigentlich kein objektorientiertes Feature dar, obwohl sie in das objektorientierte Konzept eingebunden sind. Man könnte sie sich durchaus auch als Ergänzung zu C vorstellen.

Die Idee ist, daß man bisweilen nicht über einen Member eines ganz bestimmten Klassen- bzw. Strukturobjekts, sondern nur über einen Member an sich „sprechen" will. Rein implementatorisch ist ein Member-Pointer kein wirklicher Zeiger, sondern der Offset eines bestimmten Members innerhalb einer Struktur, und konkret deklariert man solche Datentypen wie folgt:

```
struct S
{ int i, j;
  char *s;
  int k;
};

S s1, s2, s3;

int S::* pims;
```

Diese überaus komplizierte Syntax deklariert „pims" als Zeiger auf einen „int"-Member von „S". Prinzipiell werden Zeiger-auf-Member-Typen wie gewöhnliche Zeiger deklariert, nur daß eben der „*" mit einem Klassennamen qualifiziert wird. So weit, so gut, fehlt nur noch der praktische Nutzen derartiger Zeiger. Wir können die Variable „pims" jetzt mit einem Member von „S" initialisieren, aber nicht mit einem Member eines bestimmten Objekts, sondern nur mit einem Member-Namen, und der wird naturgemäß qualifiziert, und natürlich muß man, weil es sich ja angeblich um einen Zeiger handelt, dazu den unären Operator „&" verwenden:

```
void main()
{ pims = &S::i; // Fortsetzung folgt...
```

OK, jetzt enthält „pims" (falls Sie sich immer noch über diesen Namen wundern: das soll „Pointer to Int-Member of S" heißen) eine interne Repräsentierung des Offsets von „i" in „S". Jetzt können wir über „pims" auf Member von beliebigen „S"-Objekten zugreifen, und weil „pims" so initialisiert ist, erwischen wir jeweils den Member „i":

```
  s1.*pims = s2.*pims;
}
```

Diese Anweisung kopiert also einen „int"-Member von „s2" in einen Member von „s1", und da „pims" so initialisiert ist, handelt es sich jeweils um den Member „i". Dabei ist „.*" nicht eine Zusammenstellung von „." und „*", sondern ein eigenständiger binärer Operator. Analog dazu gibt es auch noch „->*", wobei

```
x->*y
```

und

```
(*x).*y
```

einander entsprechen, z. B.:

```
S *sptr = &s3;

sptr->*pims = foo;
```

Wozu das alles gut sein soll? Nun, mit Pointern auf Member kann man z. B. Member so ähnlich wie Vektor-Elemente behandeln:

```
struct S
{ int foo, bar, sucker;
};

int S::* nr(int i) // Gibt Zeiger auf den "i"-ten Member
                   // von "S" zurück
{ switch (i)
  { case 0:
      return &S::foo;
    case 1:
      return &S::bar;
    case 2:
      return &S::sucker;
    default: // Panik
  }
}

void main()
{ S s1;

  // Initialisiere "s1" wie einen Vektor:
  for (int i=0; i<3; ++i)
    s1.*nr(i) = 42 + i;

}
```

Es gibt wie bei normalen Zeigern einen ausgezeichneten Wert, nämlich „0“, den ein Member-Zeiger niemals auf „normalem“ Wege annehmen kann, d. h. es ist garantiert, daß „&x::y“ immer einen von Null verschiedenen Wert liefert. Deshalb kann man die (eigentlich numerische) Konstante „0“ benutzen, um anzuzeigen, daß ein Member-Pointer momentan auf „nichts“ zeigt, wie im folgenden (wenig sinvollen) Beispiel:

```
struct S    { char *cp1, cp2; };

char * S::* stringmemberA = &S::cp1, * S::* stringmemberB
= 0;

void f(S *sp)
{ if(stringmemberA != 0 && stringmemberB == stringmemberA)
   sp->*stringmemberA = "Oh no, Babe!";
}
```

Rein praktisch wird das in MaxonC++ so gelöst, daß ein Pointer auf Member nicht direkt durch den Offset dieses Members in der Klasse bzw. Struktur dargestellt wird, sondern um 1 größer ist, d. h. ein Zeiger auf den allerersten Member einer nicht abgeleiteten Struktur hat den Wert 1. Übrigens sehen Sie oben auch, daß man Member-Zeiger mit den üblichen Operatoren vergleichen kann. Additionsoperatoren wie bei „normalen“ Zeigern gibt es hier aber nicht - logisch, denn eine Struktur ist kein Vektor, und deshalb kann man nicht davon ausgehen, daß vor oder hinter einem Member ein weiterer Member gleichen Typs liegt.

Ansonsten wäre noch zu erwähnen, daß die objektorientierten Konzepte hier natürlich ebenfalls unterstützt werden. Im folgenden Beispiel sehen Sie, daß man einen Pointer auf einen Member einer Basisklasse direkt in einen Zeiger auf einen Member einer abgeleiteten Klasse umwandeln kann, während man für die umgekehrte Konvertierung einen Cast braucht:

```
struct Base
{ double d;
};

class Klasse: public Base
{ public:
   double e;
};

Base b;   Klasse c;

double Klasse::* kmp = &Base::d;

double Base::* bmp = (double Base::*) &Klasse::e;
```

Die Typregeln sind also genau umgekehrt wie bei Zeigern, wo man einen Zeiger auf eine abgeleitete Klasse ohne weiteres als Zeiger auf die Basisklasse verwenden kann und im umgekehrten Fall casten muß. Diese Regelung ist so aber durchaus sinnvoll, denn jede „Klasse" enthält eine „Basis", weshalb man einen Zeiger auf einen „Basis"-Member gewiß ohne Gefahr in einem „Klasse"-Objekt anwenden kann, während die Benutzung eines Zeigers auf einen „Klasse"-Member in einem „Basis"-Objekt voraussetzt, daß wir über die wahre Natur der Objekte, auf die wir diesen Zeiger später anwenden wollen, Bescheid wissen - es muß nämlich nicht nur eine „Basis", sondern sogar eine „Klasse" sein. Folgerichtig wird für diese haarige Typwandlung ein Cast verlangt.

Bei virtueller Vererbung funktionieren derartige Umwandlungen überhaupt nicht:

```
class B
{ public:
    int i;
};

class C: public virtual B
{ public:
    int j;
};

int B::* bmp = &B::i;

int C::* cmp = bmp;  // ERROR
```

Das ist, wenn man es implementatorisch sieht, auch völlig klar: Die Basisklasse „B" hat in „C" keinen festen Offset, statt dessen gibt es dort einen Zeiger auf die virtuelle Basisklasse, die weiß-der-Geier-wo liegen kann. Bei gewöhnlichen Adressumwandlungen, bei denen man es mit konkreten Objekten zu tun hat, ist das kein Problem, denn da kann man sich ja an diesen Zeigern entlanghangeln. Zeiger auf Member beziehen sich aber nicht auf ein konkretes Objekt, sondern auf eine Klasse an sich und können ohne jegliche existierende Klasseninstanz benutzt werden, also fällt das Verfolgen von Zeigern hier naturgemäß wohl aus.

Wenn Sie mich fragen, Zeiger auf Member sind die Lösung, nur hat noch niemand das Problem dazu gefunden. Es ist ein nettes kleines Feature, das man in der Praxis aber wohl von allen C++-Spielzeugen am seltensten brauchen wird.

3.5.2 Konvertierungsfuntionen

Ein Konstruktor mit einem einzelnen Argument kann bekanntlich einen beliebigen Ausdruck in ein Klassenobjekt verwandeln, etwa so:

```
class Eugen
{ int i;
  public:
    Eugen(int j)
    { i = j;
    }
};

void main()
{ int x = 1992;
  Eugen E1 = x;    // (1)
  E1 = x+2;        // (2)
}
```

Bei der Initialisierung (1) wird der Konstruktor direkt auf das Zielobjekt angewandt (d. h. „this" zeigt auf „E1"), während (2) eine gewöhnliche Wertzuweisung ist, weshalb der Compiler hier ein temporäres Objekt einführt, das er mit dem Wert „x+2" initialisiert, bevor er es in das eigentliche Zielobjekt „E1" kopiert (wie Sie sich vielleicht erinnern, unterscheidet C++ scharf zwischen Initialisierungen und Wertzuweisungen).

Weil das so toll ist (immerhin kann man ein Programm durch solche implizit aufgerufenen, anwender-definierten Typwandlungsfunktionen beliebig unübersichtlich gestalten), stellt sich natürlich die Frage, ob das vielleicht nicht auch umgekehrt geht, also eine Umwandlungsfunktion von einer Klasse nach einem beliebigen anderen Datentypen.

Im Prinzip will man also einen Konstruktor für „int" einführen, der als Argument ein Objekt der Klasse „Eugen" erwartet. Gegen diese Idee sprechen aber mindestens zwei Punkte: Erstens fällt einem dafür auf Anhieb keine anschauliche Syntax ein (Konstruktoren werden immer in der Klassendefinition, und wo zum Geier „definiert" man den Standardtypen „int") und zweitens und vor allen Dingen könnte dann jemand auf die Idee kommen, z. B. die Umwandlung zwischen „int" und „double" auf eine eigene originelle Methode zu definieren, was der Konsistenz des Sources natürlich nicht gerade zuträglich ist. Also geht man einen etwas anderen Weg: In einer Klasse kann man Konvertierungsfunktionen deklarieren, die ein Objekt dieser Klasse in einen beliebigen anderen Datentypen umwandeln und vom Compiler auch implizit, d. h. automatisch und ohne besondere Aufforderung, aufgerufen werden.

Der „Name" einer Konvertierungsfunktion besteht aus dem Wortsymbol „operator" und der Beschreibung des Typs, in den umgewandelt werden soll. Obwohl das in gewissem Sinne der Ergebnistyp der Funktion ist, darf man keinen Ergebnistypen explizit deklarieren, und natürlich hat eine Konvertierungsfunktion keine Argumente außer dem „this"-Zeiger auf ein Klassenobjekt. Ein Beispiel:

```
class HinUndHer
{ char test[20];
  public:

    HinUndHer()// Default-Konstruktor
    { test[0] = 0;
    }

    HinUndHer(const char *init)  // Noch ein Konstruktor
    : test(init)
    { }

    operator char*()  // Konvertierung nach char*
    { return test;
    }

    operator int();   // Noch eine Konvertierung

};

#include <string.h>
#include <stream.h>

HinUndHer::operator int()
{ return strlen(test);
}

void main()
{ int i;
  char *cp;

  HinUndHer h1 = "Beispiel", h2 = "Test";

  cp = h1;  // cp zeigt auf "Inhalt" von h1

  i = h2; // Länge des "Inhalts" von h2

  cout « cp « i;

}
```

Dieses Programm gibt „Beispiel4“ aus. Ohne daß man einen Cast oder einen anderen Hinweis zur Typkonvertierung angeben müßte, wandelt der Compiler „HinUndHer“-Objekte nach „int“ oder „char*“ um, indem er klammheimlich die entsprechenden Konvertierungsfunktionen aufruft.

Natürlich werden solche Funktionen (wie alle anderen Member-Funktionen auch) vererbt und können virtuell sein.

Es kann (theoretisch) vorkommen, daß der Compiler sich nicht zwischen einer Konvertierungsfunktion und einem Konstruktor entscheiden kann:

```
class A;

class B
{ public:
   B(A&);
   B();
};

class A
{ public:
   operator B();
};

void main()
{ A a;
  B b;

  b = a; // Konstruktor oder Konvertierung???
}
```

In diesem (relativ krankhaften) Fall meldet der Compiler einen entsprechenden Fehler.

Außerdem ruft er beim Versuch einer Typwandlung höchstens eine Konvertierungsfunktion auf und nicht zwei nacheinander:

```
class X
{ public:
    operator long int();
};

class Y
{ public:
   operator X();
};
```

```
void main()
{ Y y1;
  long int l1 = y1, // ERROR
  l2 = X(y1),       //  So geht's
  l3 = y1.operator X().operator long int();
}
```

In solchen Fällen muß man sich eben behelfen, indem man (wie oben auf zwei verschiedene Arten geschehen) dem Compiler ausdrücklich klar macht, wie er die Typen ineinander umzuwandeln hat.

3.5.3 Temporäre Objekte

Mehrfach war schon die Rede von „temporären Objekten", die der Compiler ohne Zutun des Programmierers erzeugt, wenn es gerade nötig ist. Eigentlich gibt es darüber fast nichts zu sagen, denn erstens ist es in hohem Maße implementationsabhängig, ob und wann ein bestimmter Compiler ein solches Objekt erzeugt und wieder löscht, und zweitens ist die ganze Angelegenheit so konzipiert, daß alles so funktioniert, wie der Programmierer es erwartet. Nur manchmal wird es tückisch, und deshalb soll hier doch etwas näher auf das Thema eingegangen werden.

Ein temporäres Objekt ist eine Variable eines Struktur- bzw. Klassentyps, die der Compiler einrichtet, um kurzfristig ein Objekt zwischenzuspeichern. Naheliegenderweise werden sie von MaxonC^{++} im automatischen Speicher, also auf dem Stack, eingerichtet, aber solche Details sind implementationsabhängig. Der Programmierer kann sich aber darauf verlassen, daß temporäre Objekte genau wie alle anderen behandelt werden, insbesondere daß sie mit den jeweils richtigen Konstruktoren initialisiert werden und daß nachher (falls vorhanden) ein Destruktor aufgerufen wird. Wann solche Destruktoraufrufe erfolgen oder der Speicherbereich, in dem das temporäre Objekt lag, wieder für andere Zwecke benutzt wird, ist aber implementationsabhängig und für den Programmierer wenig interessant. Theoretisch könnte ein Compiler temporäre Objekte jeweils mit „new" erzeugen und erst am Programmende kollektiv destruieren und ihren Speicher freigeben - ein Verfahren, das zwar alles andere als effektiv (im Laufe der Zeit können sich dabei zahllose „Leichen" ansammeln, die nichts als Speicher verbrauchen), aber durchaus erlaubt und überdies auf der sicheren Seite liegt. Keine Angst, MaxonC^{++} wählt natürlich ein wesentlich effektiveres Verfahren.

Vielleicht sollte ich Ihnen endlich verraten, wann temporäre Objekte typischerweise eingerichtet werden. Nun, im Prinzip müssen Sie immer damit rechnen. Bei einer so harmlosen Initialisierung wie

```
class C
{ // usw. public:
  C();
  C(const C&); // Copy-Konstruktor
   ~C();
};

void main()
{ C c1;
  C c2 = c1;
}
```

kann es theoretisch passieren, daß der Compiler bei der Initialisierung von „c2" durch „c1", wo eigentlich bloß der Copy-Konstruktor anzuwenden ist, 42 zusätzliche temporäre Objekte einrichtet, dann das erste mit „c1" initialisiert, das zweite mit dem ersten, das dritte mit dem zweiten, ... und schließlich „c2" mit dem 42-ten temporären Objekt initialisiert (natürlich stets unter korrekter Anwendung des Kopier-Konstruktors) und anschließend auf jedes der 42 Phantomobjekte den Destruktor anwendet. Das Beispiel ist natürlich etwas übertrieben, aber daß es „dumme" Compiler gibt, die in einer solchen Situation ein temporäres Objekt einführen, sollte man stets als Möglichkeit bedenken. Das klingt jetzt aber viel schlimmer als es ist, denn als Programmierer wird man Konstruktoren und Destruktoren rein intuitiv ohnehin so definieren, daß eine zusätzliche Initialisierung keine Probleme bereitet.

Jetzt ein Beispiel, wo jeder Compiler ein temporäres Objekt einführen muß:

```
struct S
{ int i;
};

S f();

void main()
{ int j = f().i;
}
```

Die Funktion „f" gibt, wie man so schön sagt, ein Objekt des Typs „S" zurück. In Wirklichkeit muß man ihr natürlich beim Aufruf sagen, wo sie das Ergebnis abzulegen hat, denn im Gegensatz zu skalaren Werten kann man eine Struktir nicht so ohne weiteres in einem Prozessorregister ablegen (OK OK, hier würde es natürlich zufällig klappen, aber lassen wir das). Also erzeugt der Compiler ein temporäres Objekt, in das „f" ihr Ergebnis kopieren kann, und ermöglicht dann den Zugriff auf den Member „i" dieses Objekts. Anschließend hat der Mohr seine Schuldigkeit getan und kann gehen, sprich: das Objekt kann irgendwann später gelöscht werden.

Damit wären wir auch schon (wieder) bei einem haarigen Thema, einer potenten Quelle schier unentdeckbarer Bugs und tiefster Frustration: Bisweilen existieren temporäre Objekte weniger lange, als einem lieb ist. Modifizieren wir das obige Beispiel doch ein wenig:

```
struct S
{ char str[20];
};

S f()
{ S erg = { "Saratoga" };
  return erg;
}

#include <stream.h>

void main()
{ char *cp = f().str;

  cout « cp;
}
```

Die Funktion „f" hat ein lokales Objekt namens „erg", das sauber initialisiert und beim „return" in das Zielobjekt kopiert wird. In diesem Programm ist das Zielobjekt aber wieder ein temporäres, und nun wird der „char"-Zeiger „cp" auf einen Member dieses temporären Objekts gesetzt. Die Folgen können fatal sein, denn wenn wir „cp" später benutzen, z. B. mit „cout" ausgeben, existieren das temporäre Objekt und mit ihm seine Member höchstwahrscheinlich nicht mehr! So etwas kann zu netten Überraschungen führen und man kann so etwas eigentlich nur vermeiden, wenn man mit Funktionen, die ein Objekt zurückgeben, höllisch vorsichtig ist.

Herzlichen Glückwunsch - Sie haben das Kapitel über die objektorientierten Features von C++ überstanden! Es folgen noch ein kurzes Kapitel über Möglichkeiten und Grenzen des Überladens und ein noch kürzers Kapitel zum Thema Preprozessor, und schon ist die C++-Sprachbeschreibung komplett - also keine Panik!

4. Überladen

4.1 Überladene Funktionen

4.1.1 Was man darf, und was nicht

Überladene Funktionen wurden in diesem Handbuch schon einige Male erwähnt, aber noch nicht wirklich formal (also mit allen Regeln, die dazu gehören) behandelt. Also soll in diesem Kapitel genau davon Rede sein.

Im Prinzip kann man jeden Funktionsnamen beliebig überladen, nur mit der Einschränkung, daß Funktionen mit identischen Parameterlisten keine unterschiedlichen Ergebnisse haben dürfen:

```
int f(int);

char *f(int);          // ERROR
```

Im Zusammenhang mit überladenen Funktionen sollte man zwischen den Begriffen „Funktion" und „Funktionsname" unterscheiden. Ein Funktionsname ist ein Bezeichner, der für mehrere Funktionen eines Scopes stehen kann. In Wirklichkeit gibt es also keine überladenen Funktionen, sondern überladene Namen, und das hat auch Konsequenzen:

```
class B
{ public:
  int funk(int);
};

class C: public B
{ public:
  char *funk(char*);
};

void main()
{ C c1;
  c1.funk(42);  // ERROR
}
```

Die Klasse „C" erbt zwar aus „B" eine Funktion namens „funk", aber da sie selbst einen identischen Funktionsnamen deklariert, wird „B::funk" von „C::funk" vollständig überdeckt, obwohl die Parameterlisten doch völlig unterschiedlich sind. Deshalb kennt der Compiler in „C" nicht mehr ohne weiteres „B::funk" und akzeptiert kein „int"-Argument mehr für „funk".

Es gibt noch einige andere Problemfälle von Funktionen, bei denen der Compiler nicht herausfinden kann, welche Funktion gemeint ist:

```
void g();

void g( int i = 0);

void main()
{ g(); // ERROR
}
```

Beide Funktionen können formal ohne Argumente aufgerufen werden, so daß der Compiler beim Funktionsaufruf „g()" nicht weiß, welche von beiden er nehmen soll. Prinzipiell kann ein Compiler solche Konflikte schon bei der Funktionsdeklaration erkennen und schon dort einen Fehler melden. MaxonC++ ist aber nicht ganz so schlau und entdeckt erst bei einem derartigen Funktionsaufruf, daß da etwas nicht ganz richtig sein kann. Aber das gehört schon fast in den folgenden Abschnitt, der davon handelt, nach welchen Regeln der Compiler bei überladenen Funktionsnamen anhand der Argumente die richtige Funktion herausfindet:

4.1.2 Was der Compiler davon hält

4.1.2.1 Die Hierarchie der Typwandlungen

Der einfachste Fall liegt vor, wenn eine Funktion mit genau einem Argument aufgerufen wird. Zunächst ist, auch wenn es trivial ist, zu erwähnen, daß der Compiler nur die Funktionen dieses Namens betrachtet, die auch mit einem einzelnen Argument aufgerufen werden können, d. h. solche Funktionen, die

- genau einen Parameter besitzen oder
- für alle Parameter (bzw. alle außer dem ersten) ein Default-Argument besitzen oder
- als Parameterliste nur die Ellipse „(...)" besitzen.

Nehmen wir also an, wir hätten einen geradezu üppig überladenen Funktionsnamen „f":

```
void    f();
int     f(int, int);
int     f(int);
void    f(double);
void    f(...);
short   f(char*, ...);
```

```
char f(char c1, char c2 = '?');
double f(float);
double f(struct S*);
```

Bei einem Funktionsaufruf wie „f(x)“ für einen beliebigen einzelnen Ausdruck „x“ bleiben davon folgende Funktionen im Rennen:

```
f(int);
f(double);
f(...);
f(char*, ...);
f(char c1, char c2 = '?');
f(float);
f(struct S*);
```

Die Ergebnistypen habe ich hier absichtlich weggelassen, denn sie spielen in der weiteren Analyse keine Rolle.

So weit, so cool. Bekanntlich gibt es in C++ eine Fülle von Typkonvertierungen, so daß ein Argument vom Compiler in einen passenden Parameter verschiedener Typen verwandelt werden kann. In einem Ausdruck wie „f(0)“ könnte jede der oben aufgelisteten Funktionen gemeint sein. Um hier eine gewisse Klarheit zu schaffen, gibt es in C++ eine gewisse Hitliste der Typwandlungen:

- 1. Exakte Übereinstimmung der Datentypen

Wenn es eine Funktion gibt, deren Parametertyp mit dem Argumenttyp identisch ist, macht diese natürlich das Rennen. Also würde „f(17)“ zu einem Aufruf von „f(int)“ führen. Dieser Fall ist gleichermaßen für den Compiler wie für den Programmierer der einfachste und eindeutigste. Wenn man sich gerade über die Typregeln nicht sicher ist, kann man natürlich immer mit einer expliziten Typwandlung, etwa „f(int(x))“, eine solche exakte Übereinstimmung erzwingen und ist so vor unliebsamen Überrschungen geschützt.

- 2. Triviale Typumwandlungen

Die folgenden Umwandlungen gelten für jeden Datentypen „T“ als trivial:

```
T         ->   T&
T&        ->   T
T[x]      ->   T*
T         ->   const T    // x
T         ->   volatile T // x
T*        ->   const T*   // x
T*        ->   volatile T*     // x
T(arg)    ->   (*T)(arg)
```

Die letzte Regel soll die Umwandlung eines Funktionsnamens in einen Zeiger auf eine Funktion symbolisieren.

Eine Typwandlung, die ausschließlich diesen Regeln folgt, gilt als trivial, auch dann, wenn sie aus einer Folge solcher Konvertierungen besteht (z. B. T& -> const T).

Es gibt hier noch eine Unterhierarchie: Die Umwandlungen, die oben mit einem „x" gekennzeichnet wurden, gelten als schlechter als die übrigen. Entsprechend sind auch Folgen trivialer Konvertierungen, die diese Konvertierungen enthalten, schlechter als solche, die ausschließlich aus den anderen bestehen.

- 3. Aufsteigende Konvertierungen

Wie Sie sich vielleicht erinnern, rechnet C++ im wesentlichen mit 6 verschiedenen Datentypen, das sind

```
int   unsigned int   long   unsigned long   double   long   double
```

Es steht einer Implementierung frei, auch in „float" Berechnungen auszuführen, wenn alle Operanden eines Operators „float" sind, und MaxonC++ tut dies auch. Ansonsten braucht sich der Compiler mit kleineren Typen gar nicht abzugeben und wandelt bei Berechnungen Ausdrücke der „kleinen" Typen „short" und „char" (jeweils sowohl vorzeichenlos als auch vorzeichenbehaftet) in „int" um. Wenn eine Implementierung die oben erwähnten Berechnungen im „float"-GBereich nicht bietet, wandelt sie „floats" nach „double" um.

Damit man diesen Mechanismus auch bei überladenen Funktionen benutzen kann, gelten die Umwandlungen

```
char   unsigned char   signed char   -> int   short   unsigned
short

sowie

float -> double
```

als „aufsteigende Umwandlungen" (neudeutsch „Promotions") und stellen die dritte Stufe der Konvertierungshierarchie von C++ dar. Folgen von Typumwandlungen, die nur solche aufsteigenden Umwandlungen und eventuell auch die oben erwähnten trivialen Konvertierungen enthalten, sind besser als alle nachfolgend aufgelisteten Umwandlungen.

Was Sie als Programmierer davon haben? Nun, um eine numerische Funktion, z. B. den absoluten Betrag, auf allen numerischen Datentypen so zu definieren, daß es für jedes numerische Argument genau eine passende Funktion gibt, reicht es, wenn Sie sechs verschiedene Funktionen definieren, z. B. so:

```
int Abs(int);
unsigned Abs(unsigned);
long Abs(long);
unsigned long Abs(unsigned long);
double Abs(double);
long double Abs(long double);
```

In MaxonC++ sind natürlich „int" und „long" praktisch identisch und diese Unterschiedung ist nur für den Compiler von Bedeutung. Wenn Sie die speziellen Features von MaxonC++ nutzen wollen, kämen auch noch

```
long long Abs(long long);
unsigned long long Abs(unsigned long long);
```

dazu, aber das ist bekanntlich nicht Standard.

Natürlich haben Sie auch noch die Möglichkeit, die „Abs"-Funktion für weitere Parametertypen, z. B. „short", zu definieren, ohne die Eindeutigkeit von Funktionsaufrufen zu beeinträchtigen: Bei „Abs" mit einem „short"-Argument würde eben jene zusätzliche Funktion aufgerufen, während „unsigned short"- oder „char"-Argumente weiter nach „int" expandiert würden.

Umgekehrt gibt es aber Probleme, wenn Sie einen der sechs Typen auslassen: Wenn z. B. „Abs(unsigned)" nicht deklariert wird, weiß der Compiler nicht, was er tun soll, wenn „Abs" mit einem „unsigned"-Argument aufgerufen wird, denn sowohl die Umwandlung nach „int" als auch die nach „long" oder „unsigned long" fällt schon in die nachfolgende Kategorie (nebenbei bemerkt ist es natürlich ziemlich schwachsinnig, die Betragsfunktion auf vorzeichenlosen Datentypen zu definieren, aber es sollte ja auch bloß ein Beispiel sein).

- 4. Standard-Konvertierungen

Diese Kategorie umfaßt alle vordefinierten Typumwandlungen, die oben noch nicht erwähnt wurden, nämlich

- alle sonstigen Umwandlungen zwischen numerischen Datentypen, beispielsweise von „double" nach „signed char" und was Ihnen noch so alles an möglichen Kombinationen einfällt,

- die Umwandlung der Konstanten „0“ in einen Zeiger,

- die Konvertierung eines beliebigen Zeigertyps nach „void*“,

- Umwandlungen eines Zeigers auf eine Klasse in einen Zeiger auf eine direkte oder indirekte Basisklasse davon,

- die analoge Umwandlung zwischen Referenzen,

- Konvertierungen von Pointern auf Member in einen jeweils passenden Memberzeiger einer abgeleiteten Klasse.

- die Umwandlungen von einer abgeleiteten Klasse in eine ihrer Basisklassen.

Bei den Zeigerumwandlungen gibt es aber noch eine Unter-Hierarchie: Wenn von einer Klasse „A“ eine Klasse „B“ und davon wiederum eine Klasse „C“ abgeleitet wurde, ist die Umwandlung von „ C* “ nach „ B* “ besser als die von „ C* “ nach „ B* “. Hier geht also die „Tiefe“ einer solchen Umwandlung in ihre Güte mit ein. Für Referenzen auf Klassen sowie für Klassen selbst gilt analoges, und bei Zeigern auf Member ist es entsprechend umgekehrt, d. h. hier ist „A::* -> B::*“ besser als „A::* -> C::*“.

Damit wären alle Typkonvertierungen erwähnt, die der Compiler „von Natur aus“ kennt. Typwandlungen, die in die Kategorie der Standard-Konveriterungen fallen, sind schlechter als die davor beschriebenen Konvertierungen, aber stets besser als vom Anwender (also dem Programmierer (also Ihnen)) deklarierte Konvertierungsfunktionen:

- 5. Anwenderdefinierte Konvertierungen

Der Programmierer kann bekanntlich auch selbst Regeln zur Typwandlung deklarieren, und zwar in Form von Konstruktoren und Konvertierungsfunktionen. Der Compiler berücksichtigt bei Gelegenheit derartige Typwandlungen und führt bei Bedarf sowohl vorher als auch nachher Standard-Konvertierungen durch. Er versucht aber nicht, mehrere anwenderdefinierte Konvertierungen nacheinander auszuführen.

Typwandlungen, die anwenderdefinierte Konvertierungen benutzen, sind schlechter als alle anderen, wobei es in der Hierarchie keinen Unterschied zwischen Konstruktoren und Konvertierungsfunktionen gibt.

Vielleicht fühlen Sie sich jetzt diskriminiert und zurückgesetzt, weil der Compiler „Ihre“ Typkonvertierungen für „schlechter“ als seine eigenen hält, obwohl Sie doch sicherlich intelligenter als so ein dummer Compiler sind. Das ist aber nicht wirklich so, denn in den meisten Fällen treten Sie hier gewissermaßen „außer Konkurrenz“ an: Mit einem Konstruktor können Sie festlegen, wie ein beliebiger (z. B. numerischer) Wert in ein Klassenobjekt umgewandelt wird, und Konvertierungsfunktionen behandeln die umge-

kehrte Wandlung eines Klassenobjekts in irgendeinen anderen Ausdruck. Für solche Umwandlungen gibt es aber bekanntlich keine Standard-Konvertierungen.

Außerdem tröstet es Sie vielleicht, daß es bei Funktionsaufrufen auch noch eine „Konvertierung" gibt, die noch schlechter als ihre anwenderdefinierten ist:

- 6. Funktionsaufruf mit Ellipse

Die Übergabe eines Arguments als „Ellipsen-Parameter" ist schlechter als alle anderen Parameterübergaben, auch wenn diese mit noch so schlechten Typkonvertierungen erfolgen. Das heißt in der Praxis, daß eine Funktion „f(...)" nur aufgerufen wird, wenn es wirklich keine andere Funktion namens „f" gibt, auf die das Argument irgendwie paßt.

Oft besteht eine Typkonvertierung aus mehreren Schritten, z. B. erst von „int" nach „int&" und dann nach „const int&", was jeweils unter die Rubrik „triviale Umwandlung" fällt. Wenn der Compiler bewerten soll, welche von zwei Typkonvertierungen die Bessere ist, zählt ausschließlich der jeweils schlechteste Einzelschritt. Das hat bisweilen unangenehme Konsequenzen: Nehmen wir an, Sie versehen eine Klasse mit einer Konvertierungsfunktion nach „const char*".
Da man Daten dieses Typs normalerweise mit „cout" ausgeben kann, liegt es nahe, das auch direkt mit einem Klassenobjekt zu tun:

```
#include <stream.h>

class C
{ char Inhalt[20];
  public:
    operator const char*();
};

void main()
{ C c1;
  cout « c1; // ERROR: Ambiguous function call
}
```

Wieso das? Nun, mit „cout" kann man zwar „const char*"-Objekte, also Strings, ausgeben. Außerdem gibt es aber noch (neben vielen anderen, die uns hier nicht interessieren) die Möglichkeit, ein „const void*" auszugeben, nämlich die hexadezimale Adresse eines Objekts. Der Compiler hat nun zwei Möglichkeiten, „c1" in ein Argument für „cout «" umzuwandeln: Einerseits die kanonische, im obigen Beispiel vom Programmierer wohl gemeinte einfache Umwandlung

```
C -> const char*
```

und andererseits die Konvertierungsfolge

```
C -> const char*-> const void*
```

Überraschung, Überraschung: Für den Compiler sind diese Konvertierungen gleichwertig, denn es zählt ja jeweils nur der schlechteste Schritt, hier also die anwenderdefinierte Konvertierungsfunktion. Eine nachfolgende Standardkonvertierung, z. B. von „const char*" nach „const void*", fällt dann nicht mehr ins Gewicht.

Es ist aber durchaus möglich, Klassen bequem und direkt auszugeben, indem man den Operator „«" überlädt. Dazu lesen Sie in 4.2.2.7 mehr.

4.1.2.2 Matching von Funktionsargumenten

Aus der obigen Hierarchie ergibt sich direkt, welche Funktion aufgerufen wird, wenn nur ein Argument übergeben wird: Der Compiler sucht sich aus allen Funktionen dieses Namens gerade die heraus, die nach dieser Hierarchie am besten paßt - oder sagen wir besser „passen", denn es kann durchaus auch mehrere Funktionen geben, die „gleich gut" sind:

```
void f(int);
void f(double);

void main()
{ int i;
  short s;
  unsigned u;

  f(i);  // OK, exakte Übereinstimmung mit "int"

  f(s);  // OK: aufsteigende Konvertierung nach "int", aber
         // nur Standard-Konvertierung nach "double"

  f(u);// ERROR: ambiguous function call!
       // unsigned -> int und unsigned -> double sind
       // jeweils Standard-Konvertierungen, also gleich
       // schlecht!
}
```

So weit, so gut - aber was ist, wenn einer Funktion mehrere Argumente übergeben werden? Auch dann haben wir es mit einer ziemlich einfachen Regel zu tun, die man am besten mit einer Mengen-Terminologie (hat nichts mit Terminatoren zu tun) ausdrückt. Hier betrachtet der Compiler zunächst jeden Parameter einzeln und ermittelt dabei genau wie zuvor die Funktionen, bei denen dieses einzelne Argument am besten paßt. Das Ergebnis ist also eine Memge von Funktionen. Wenn er das für jedes Argument getan hat, bildet er die Schnittmenge über alle diese Mengen.

Oder, um es anders zu formulieren: Die Funktion, die insgesamt am besten paßt, ist gerade die, die bei jedem einzelnen Parameter zu den besten gehört. Es kann hier mehrere dieser Funktionen geben - oder auch gar keine.

Betrachten wir ein einfaches Beispiel mit zwei Funktionen, die jeweils drei Parameter besitzen:

```
int d(int, int, int);          // Funktion (a)
long d(long, double, double); // Funktion (b)

void main()
{ int i1, i2, i3;
  char c1;
  unsigned u1, u2, u3;
  long l1, l2, l3;
  double d1, d2, d3;
  float f1, f2, f3;

  d (d1, c1, l1); // (1) OK

  d (i1, d2, u3); // (2) ERROR

  d (f1, f2, f3); // (3) OK

  d (u1, l2, l3); // (4) ERROR
}
```

Zwei der Funktionsaufrufe sind für den Compiler eindeutig und die beiden anderen eben nicht. Gehen wir doch einmal die vier Beispiele langsam und zum Mitdenken durch:

(1) Als erstes Argument wird ein „double" übergeben, und sowohl nach „int" als auch nach „long" haben wir es mit einer Standard-Konvertierung zu tun. Also ist die Funktionsmenge für das erste Argument {a, b}. Das zweite Argument vom Typ „char" kann mit einer aufsteigenden Konvertierung in ein „int" verwandelt werden, was besser als die Standard-Konvertierung nach „double" ist. Also ist die zweite Menge {a}, während wir beim letzten Argument wieder zwei Standard-Konvertierungen gleicher Güte vorliegen haben (Menge {a, b}). Die Schnittmenge von {a, b}, {a} und {a, b} ist bekanntlich {a}, und so haben wir hier einen eindeutigen Aufruf von Funktion (a) vorliegen.

(2) Das erste Argument paßt exakt zu Funktion (a), während bei (b) eine Standard-Konvertierung nötig wäre. Also ist die beste Menge hier offensichtlich {a}. Beim folgenden „double"-Argument ist es genau umgekehrt: Es paßt exakt zur Funktion (b), so daß die Funktionsmenge {b} herauskommt. Das dritte Argument paßt wiederum bei beiden Funktionen gleich schlecht (Standard-Konvertierung), wes-

halb wir hier {a, b} notieren. Die Schnittemenge von {a} * {b} * {a, b} ist aber leer - es gibt also keine Funktion, die bei allen Argumenten zu den besten gehört.

(3) Beim ersten Argument haben wir es mit zwei Standard-Konvertierungen zu tun (Menge {a, b}), während die beiden folgenden Argumente jeweils mit einer aufsteigenden Typwandlung in „double" verwandelt werden können (zweimal Menge {b}). Also ist (b) hier klarer Sieger.

(4) Bei allen drei Argumenten ist jeweils für beide Funktionen eine Standard-Konvertierung anzuwenden, weshalb für alle drei Mengen {a, b} herauskommt. Der Durchschnitt über dreimal diese Menge ist eben {a, b}, und so haben wir hier gleich zwei Funktionen die überall zu den am besten passenden gehören. Also ist dieser Funktionsaufruf mehrdeutig und deshalb ein ERROR.

Solche Analysen sind natürlich ziemlich krankhaft, aber das ist bei weitem nicht so schlimm, wie es vielleicht scheinen mag: In der Praxis kann man schließlich immer durch explizite Typkonvertierungen eindeutig und leicht ersichtlich vorgeben, was man meint.

4.2 Operatoren

4.2.1 Allgemeine Operatoren

4.2.1.1 Allgemeines über allgemeine Operatoren - allgemeiner geht's kaum

Das mit den überladenen Funktionen ist ja ganz erbaulich, aber vielleicht fragen Sie sich trotzdem (zu Recht), was das Ganze soll. Man könnte in der Tat ohne überladene Funktionen auskommen, wenn, wenn es in C nicht schon ohnehin überladene Funktionen gäbe: Fast jeder Operator symbolisiert im Prinzip eine ganze Menge von ein- oder zweistelligen Funktionen. Beispielsweise steht das binäre „+" für nicht weniger als sechs oder sieben verschiedene Operationen, nämlich Addition von jeweils zwei „int"-, „unsigned"-, „long"-, „unsigned long"-, „double"- oder „long double"-Werten sowie in einigen Implementationen auch noch die „float"-Addition (und in MaxonC++ obendrein auch noch Additionen auf „long long" sowie „unsigned long long"). Außer diesen numerischen Additonen gibt es natürlich auch noch die Addition eines Pointers mit einer ganzen Zahl. Der Compiler braucht also sowieso Regeln, um aus den Argumenten auf die gewünschte Funktion zu schließen, und so sind überladene Funktionen gewissermaßen nur ein Feature, das mit den ohnehin vorhandenen überladenen Operatoren eng verwandt ist und deshalb leicht in den Sprachstandard integriert werden konnte.

Der einzige Unterschied ist bis jetzt, daß Operatoren vom Compiler fest vorgegeben sind, während Funktionen vom Anwender definiert werden. Aber genau das ändert sich mit diesem Abschnitt: Ab sofort ist ein „+" nicht meht bloß eine Addition, wie sie der Standard definiert - jetzt werden wir kreativ.

So gut wie jeder ein- oder zweistellige Operator kann vom Programmierer noch weiter überladen werden, als er es ohnehin schon ist. Sie kennen schon ein sehr nützliches Beispiel dafür: Die Operatoren „«" und „»", mit denen man gewöhnlich Bits durch die Gegend schiebt, werden im Standard-Include „stream.h" auf den Klassen „istream" bzw. „ostream" überladen und wuchten plötzlich nicht mehr bloß binäre Daten durch den Speicher, sonderm bewirken höchst komplexe Ein- und Ausgabe-Operationen.

Aber bevor Sie in unangemessene Ekstase geraten, sei Ihre Euphorie ein wenig gedämpft, denn es gibt einige nicht unbedeutende Einschränkungen:

- Man kann keine völlig neuen Operatoren definieren, sondern ausschließlich den vorhandenen Operatoren zusätzliche Bedeutungen verleihen. So mancher würde z. B. einen Operator namens „**" oder „pot" für Potenzierungen einführen, aber das geht in C++ nicht (sagen wir besser: nicht einmal in C++).

- Weder Vorrang noch Bindung von Operatoren können verändert werden. Sie können also „+" und „*" soviel überladen, wie Sie wollen, aber Punktrechnung geht weiter vor Strichrechnung und die Bindung erfolgt dabei von links nach rechts, basta.

- Die Operatoren bleiben ein- oder zweistellig (d. h. sie besitzen ein oder zwei Parameter). Bei anwender-definierten Überladungen muß mindestens einer dieser Parameter ein Klassentyp oder eine Referenz auf eine Klasse sein. Das ist aber halb so schlimm, denn so vermeidet man, daß die von Ihnen definierten Operatoren mit den Standard-Operatoren kollidieren; sonst könnte z. B. ein Witzbold auf die Idee kommen, „/" mit einen „double" als linkem und „unsigned short" als rechtem Parameter eine ganz andere Bedeutung zu geben (zur Erinnerung: normalerweise wird ein solcher Ausdruck als Division in „double" ausgewertet).

Es war die Rede davon, daß man fast alle Operatoren überladen kann. Hier ist eine Liste der überladbaren Operatoren:

```
+ - * / % ^ & |
~ ! = < > += -= *=
/= %= ^= &= |= « » »= «= == != <= >= && || ++ -- , -> ->*
() [ ] new delete
```

Dabei steht „()" für Funktionsaufrufe und „[]" für Indizierungen. Die Operatoren „+", „-", „*" und „&" können gleichermaßen als einstellige wie als zweistellige Funktione überladen werden, und „++" sowie „--" können als Postfix- oder Präfix-Operator unterschiedliche Bedeutungen erhalten.

Die Operatoren „." und „.*" können nicht überladen werden, da sie auf Klassenausdrücken (zumindest auf der linken Seite) bereits eine vordefinierte Bedeutung haben umd man hier Kollisionen und Inkonsistenzen vermeiden will. Das Symbol „::", das zur Compilezeit Scopes auflöst, ist nur im entferntesten Sinne als Operator aufzufassen und kann folglich ebenfalls nicht überladen werden, ebenso wenig wie der bedingte Ausdruck „:?", der als dreistelliger Operator (Bedingung, „if"- und „else"-Teil) ein wenig aus der Reihe fällt. Es versteht sich fast von selbst, daß das „#" und „##", die ausschließlich vom Preprozessor benutzt werden, gleichfalls nicht überladen werden können. Sonst ist in C++ aber alles überladbar, was nicht niet- und nagelfest ist, und wahrscheinlich sind Sie schon höllisch neugierig, wie man z. B. einen Funktionsaufruf überlädt, aber zuerst befassen wir uns hier mit eher gewöhnlichen Operatoren:

4.2.1.2 Wie man so etwas deklariert

Ein selbstdefinierter Operator ist zunächst einmal eine Funktion, die einen Namen hat. Die Funktion zu einem Operator „x" heißt „operator x" und wird auch als solche deklariert, z. B.

```
class X
{ public:
    int i;
};

int operator + (X x, int j)
{ return x.i + j + 42;
}
```

Und schon gibt es eine neue Addition, und zwar mit einem „X" als linkem und einem „int" als rechtem Operanden. Diese Operation ist jetzt aber nicht kommutativ, d. h. bei einer Addition „int + X" würde der Compiler diese neue Operation nicht aufrufen. Deshalb empfiehlt es sich in solchen Fällen normalerweise, die umgekehrte Operation auch noch zu deklarieren und definieren:

```
int operator + (int j, X x)
{ return x.i + j + 42;
}
```

Bei Operatorfunktionen muß die Anzahl der Parameter mit der Argumentanzahl der jeweiligen Standard-Operatoren übereinstimmen. Beispielsweise kann „operator -" wahlweise ein oder zwei Parameter haben (unäres bzw. binäres Minus!), während „/" immer genau zwei Parameter haben muß. Natürlich sind dann Default-Argumente und die Ellipse „..." verboten.

Davon einmal abgesehen, sind Operatoren ganz gewöhnliche Funktionen, die bloß etwas seltsam heißen („operator xxx" statt eines Bezeichners) und auf eine noch seltsamere Art

und Weise aufgerufen werden (siehe auch im nächsten Abschnitt). Sie können „friend" einer Klasse sein - oder auch ein Member:

```
class B
{ double d;
  public:
    B &operator += (double);
};

B& B::operator += (double x)
{ d += x;
  return *this;
}
```

In solchen Fällen dient der „this"-Zeiger stets als linker Operand, so daß der zweistellige Operator „+=" hier nur noch einen weiteren Parameter haben darf (und muß). Entsprechend dürfen einstellige Operatoren, wenn sie als Member deklariert werden, gar keine weiteren Parameter besitzen.

Wenn ein Operator als Member-Funktion einer Klasse deklariert wird, muß beim Funktionsaufruf der linke Operand naturgemäß ein L-Wert sein, auf den dann „this" zeigt (notfalls führt der Compiler aber auch ein temporäres Objekt ein). Es ist ziemlich egal, ob ein Operator als Member-Funktion oder auf Dateiebene deklariert wird, zumal auch die Scope-Regeln hier keinen direkten Einfluß haben (globale Operator-Funktionen überdecken gleichnamige Member nicht - das wäre ja auch fatal, denn es gibt jeden Operator sowieso schon als global vordefinierten Standard-Operator, so daß Member-Operator-Funktionen dann stets im Scope der Klasse versteckt und von außen überdeckt wären). Im Prinzip ist so etwas ganz einfach Geschmackssache, einmal davon abgesehen, daß einige Operatoren (z. B. „=") ausschließlich als Member-Funktion deklariert werden dürfen. Solche Sonderregelungen werden in Abschnitt 4.2.2 beschrieben.

4.2.1.3 Und wie man es benutzt

Da ein Operator eine (fast) normale Funktion ist, kann man ihn auch so aufrufen:

```
#include <stream.h>

class A
{ public:
    A operator ! ();
};

operator & (A& a1, A& a2);
```

```
void main()
{ A a, b;
  a.operator!();
  operator & (a, b);

  // in <stream.h> wird "«" auf Objekten der Klasse "ostream",
  // zu denen auch "cout" gehört, überladen, also könen wir
  // folgendermaßen Daten ausgeben:
  cout.operator « ("Hello, World!\n");

  // Allerdings ist jener Operator eine Member-Funktion, so daß
  // Folgendes nicht erlaubt ist:
  operator « (cout, "So geht's auch.\n"); // ERROR!
}
```

Das ist aber natürlich nicht der Sinn der Sache. Wesentlicn eleganter geht das nämlich so:

```
class A
{ public:
    A operator ! ();
};

A operator & (A& a1, A& a2);

void main()
{ A a, b, c;

  b = !a;
  c = a & b;
}
```

Wenn Sie einen Operator benutzen, wendet der Compiler zum Auflösen der Überladung dieselben Regeln an, wie man sie von gewöhnlichen Funktionen kennt (vgl. 4.1.2), so daß es auch hier zu nicht eindeutigen Funktionsaufrufen kommen kann. Selbstdefinierte Operatoren sind gleichrangig mit vordefinierten.

4.2.2 Spezielle Operatoren

4.2.2.1 Der Zuweisungs-Operator „="

Das Kopieren von Klassenobjekten ist ein Vorgang ganz besonderer Art, weshalb der Copy-Konstruktor auch eine gewisse Sonderstellung einnimmt (z. B. wird er als einziger Konstruktor vom System defaultmäßig definiert). Allerdings wird dieser Konstruktor naturgemäß nur zur Initialisierung von Objekten eingesetzt, während er auf gewöhnliche Wertzuweisungen zwischen Objekten keinen Einfluß hat. Diese Lücke schließt man mit der Möglichkeit, den Operator „=" zu überladen.

Der Operator „=" darf ausschließlich als Member-Funktion definiert werden, z. B. wie im folgenden Beispiel, das ansatzweise (und auch nicht sehr geschickt) eine kleine String-Bibliothek implementiert:

```
#include <string.h>

class String
{ char *s;
  public:
    String &operator = (const String&);
    String &operator = (char*);
};

String &String::operator =(char* init)
{ delete s; // alten String-Inhalt löschen
  s = init;
  return *this;
}

String &String::operator = (const String &str)
{ delete s;           // Alten String löschen
  s = new char[strlen(str.s)+1]; // Neuen allozieren
  strcpy(s, str.s);   // Stringinhalt kopieren
  return *this;       // Referenz zurückgeben
}

void main()
{ String s1, s2;

  s1 = "Test";  // Entspricht s1.operator =("Test");
  s2 = s1;       // Entspricht s1.operator =(s1);
}
```

Normalerweise ist es das Sinnvollste, als Ergebnis eines Zuweisungsoperators das soeben initialisierte Objekt (also „*this") bzw. eine Referenz darauf zurückzugeben, um

so die Semantik einer normalen Wertzuweisung zu imitieren. Verlangt wird das aber keineswegs.

Der Zuweisungsoperator mit einer Referenz auf ein Klassenelement als Argument nimmt eine Sonderstellung unter den überladbaren Operatoren ein: Als einziger besitzt er eine defaultmäßige Definition. Zu jeder Klasse K gibt es einen Operator (oder zumindest so etwas ähnliches)

```
K &K::operator = (const K &),
```

der Klassenobjekte Member für Member kopiert. Der Programmierer hat dann die Möglichkeit, diesen Operator selbst zu deklarieren und dann zu definieren (wobei er auch das „const" weglassen darf). Dies ist ein bedeutender Unterschied zu allen anderen Operatoren, denn mit „+", „*" oder „+=" kann man ja lediglich Operationen, die es normalerweise nicht gibt, auf Klassen definieren.

Diese besondere Rolle hat einige Konsequenzen: Der Operator „=" mit einer Referenz auf seine eigene Klasse ist der einzige Operator, der wirklich vererbt und in abgeleiteten Klassen bei Bedarf auch automatisch deklariert wird. Dies geschieht völlig analog zur Behandlung von Copy-Konstruktoren (3.3.4.2):

```
class Base
{ public:
    Base();
    Base(const Base&);
    Base &operator = (const Base&);
};

class Derived : Base
{ public:
    Derived();
};

void main()
{ Derived d1, d2(d1);
  d1 = d2;
}
```

In der Klasse „Derived" definiert der Compiler automatisch einen Kopier-Konstruktor

```
Derived::Derived(const Derived &)
```

sowie einen Zuweisungsoperator

```
Derived &Derived::operator = (const Derived &)
```

die jeweils zuerst den Kopier-Konstruktor bzw. Zuweisungsoperator der Basisklasse aufrufen und anschließend den Rest der Klasse ganz normal kopieren.

Von Compiler generierte Zuweisungsoperatoren haben den Zugriffsmodus „public" - auch das ist genau wie bei den Konstruktoren.

Auch wenn „x += y" bei Standard-Datentypen mit „x = x + (y)" identisch ist, hat die Definition eines eigenen „="-Operators auf einer Klasse keineswegs zur Folge, daß auch „+=" oder ähnliche Operatoren automatisch aufgerufen werden können. Wenn man solche Operatoren auf einer Klasse definiert haben möchte, muß man das schon selbst tun. Diese müssen übrigens nicht Member einer Klasse sein:

```
class C    {      // ...    };

C &operator = (C&, const C&);          // ERROR

C &operator +=(C&, const C&);          // OK
```

Warum aber muß „=" stets als Member deklariert werden? Nun, auf diese Weise stellt der Compiler sicher, daß die Gestalt des Zuweisungsoperators (für den es ja auch eine Default-Bedeutung gibt) fest steht, sobald die Klassendefinition abgeschlossen ist. Folgender Code wäre einfach zu grausam:

```
class Err   {     // ...   };

Err e1, e2;

void Falsch()
{ e1 = e2; // Standard-Zuweisung
}

Err &operator = (const Err&, Err&); // ERROR

void TotalFalsch()
{ e1 = e2;    // Selbstdefinierte Zuweisung
}
```

Da der Compiler bekanntlich nicht hellsehen kann, würde er die erste Zuweisung für ein Standard-"=" (also memberweises Kopieren) halten, während die zweite offensichtlich ein Aufruf der anwenderdefinierten Funktion ist. Entweder müßte der Compiler also einen unangemessenen Aufwand treiben, um solche Fälle als Fehler abzufangen bzw. bereits gelesene Zuweisungen nachträglich noch einmal zu betrachten (das gefällt dem Compilerbauer gar nicht), oder die Zuweisungsoperation wäre inkonsistent (das ist nichts für Ästheten). Also ist der Zwang, „=" als Member zu deklarieren, die sinnvollste Lösung. Dagegen stellt sich dieses Problem bei „+=" oder „*=" nicht, denn diese Operatoren haben bekanntlich keine vordefinierte Bedeutung auf Klassen.

4.2.2.2 „new" und „delete"

4.2.2.2.1 Selbstgeschnitzte Speicherverwaltung

Bei „new" und „delete" werden gewöhnlich fest vordefinierte Funktionen aus den Bibliotheken des Compilersystems aufgerufen. Oft ist diese Speicherverwaltung aber nicht effektiv genug: Möglicherweise möchte man Objekte einer bestimmten Klasse zunächst immer nur allozieren und dann am Programmende auf einen Schlag vollständig löschen. Wenn man so etwas sowohl auf Laufzeit als auch auf Speicherbedarf optimieren möchte, stellt man schnell fest, daß man mit der vollständigen Heap-Verwaltung eigentlich mit Kanonen auf Spatzen schießt. Schließlich müssen das Laufzeitsystem und das Betriebssystem des Rechners gleichermaßen umfangreiche Listenstrukturen verwalten, um sich zu merken, welchen Speicher das Programm reserviert hat und wo noch etwas frei ist. Das kostet einen nicht unerheblichen Overhead an Speicherplatz und Laufzeit.

Eine beliebte Lösung für das oben angedeutete Problem ist, sich einmal einen gewissen Speicherblock (sagen wir: 10 KByte) mit den Standard-Funktionen zu allozieren und in diesem Speicherbereich dann alle dynamisch erzeugten Objekte nacheinander abzulegen, bis er voll ist - und dann richtet man eben noch einen solchen Block ein. Da zwischendurch kein „delete" benutzt wird, braucht man keine wirkliche Heap-Verwaltung und kann deshalb sehr schnell Datenobjekte allozieren und braucht auch praktisch keinen Speicher-Overhead. Der MaxonC++ Compiler verwaltet fast alle bei einem Übersetzungsvorgang anfallenden Daten auf diese Weise, was dann auch nicht unerheblich zu der beeindruckenden Übersetzungsgeschwindigkeit beiträgt.

Auch sonst lassen sich sicher noch andere Gelegenheiten denken, bei denen die allgemeine Standard-Speicherverwaltung den aktuellen Erfordernissen nicht genügt. Lange Rede, schwacher Sinn: C++ unterstützt anwenderdefinierte Speicherverwaltungen durch Überladen der Operatoren „new" und „delete".

Die Operatoren „new" und „delete" können als Member von Klassen deklariert werden, wodurch dann Objekte dieser Klasse (und aller abgeleiteten Klassen) mit diesen Operatoren alloziert und freigegeben werden. Dabei gibt es aber das kleine Problem, daß ein Programm bei einem „new"-Aufruf erst Speicher allozieren muß und erst dann den Konstruktor aufrufen kann, und umgekehrt den Speicher erst freigeben kann, nachdem der Destruktor ausgeführt wurde. Also operieren Speicherverwaltungsfunktionen nicht auf einem Objekt und sind deshalb grundsätzlich statische Member (d. h. ohne „this"-Zeiger), auch wenn man das nicht explizit so deklariert. Wie alle statischen Funktionen können „new" und „delete" folglich auch nicht virtuell sein.

4.2.2.2.2 Überladen von „new"

Im Include-File „<stddef.h>" steht ein „typedef", das einen numerischen Datentypen unter dem Namen „size_t" definiert. Bei MaxonC++ handelt es sich dabei um den Typen „unsigned int". Ein „new"-Operator muß natürlich wissen, wieviel Speicher er reservieren soll, und so muß der erste Parameter immer jenen Typen „size_t" haben:

```
#include <stddef.h>

class Beispiel
{ public:
    void *operator new(size_t Groesse);
    void *operator new(size_t Groesse, int Nochwas);
};
```

Der Ergebnistyp eines „new"-Operators muß stets „void*" sein, und das hat auch seine Richtigkeit, denn auch wenn der Operator benutzt wird, um Speicher für ein Objekt der Klasse „Beispiel" zu reservieren, und einen Zeiger auf den allozierten Speicher (bzw. 0 bei Fehler) zurückgeben muß, zeigt das Ergebnis einfach nur auf einen Speicherbereich - ein „Beispiel"-Objekt wird daraus erst, wenn der entsprechende Konstruktor aufgerufen wird.

Bei einem Ausdruck wie

```
Beispiel *bp = new Beispiel;
```

ruft der Compiler zuerst die Funktion

```
Beispiel::operator new(sizeof(Beispiel))
```

auf und hofft, daß diese einen von 0 verschiedenen Zeiger zurückgibt. Das Ergebnis weist er dann an „bp" zu und ruft gegebenenfalls einen Konstruktor auf dem entsprechenden Speicherbereich auf. Ein selbstdefiniertes „new" und „delete" braucht sich also nur um die Speicherverwaltung zu kümmern, während die Sache mit den Konstruktoren und Destruktoren weiter Sache des Compilers bleibt.

Bestimmt sind Sie jetzt begierig, zu erfahren, was es oben mit dem zweiten „new"-Operator (der mit dem zusätzlichen Parameter) auf sich hat. Man darf unmittelbar hinter einem „new" eine Argumentliste angeben, die dann direkt als zusätzlicher Parameter (der erste Parameter ist nach wie vor die Elementgröße) an eine passende „new"-Funktion übergeben werden, z. B. so:

```
#include <stream.h>
#include <stddef.h>
#include <stdlib.h>

class C
{ int i, j, k;
  public:
    void *operator new(size_t, char*);
    void *operator new(size_t, int, int);
    C();
};

void *C::operator new(size_t Size, char *Message)
{ cout « Message « "\n";
  return ::new char[Size];  // So greift man gezielt
                            // auf das globale "new" zu
}

void *C::operator new(size_t Size, int x, int y)
{ cout « "Mit den Argumenten "« x« " und "« y « " werden "
  « Size « " Bytes alloziert.\n";
  return ::operator new (Size); // So geht's auch
}

C::C()
{ cout « "Objekt " « this « " wird initialisiert.\n";
  i = j = k = 0;
}

void main()
{ C *cp1 = new("And now to something completely different")
  C, *cp2 = new(17,4)C;
}
```

Die beiden „new"-Anweisungen im Hauptprogramm übergeben zusätzliche Parameter an den „new"-Operator, so daß die entsprechenden Funktionen aus der Klasse „C" aufgerufen werden. Diese geben hier lediglich ein paar Meldungen aus und reservieren dann Speicher mit dem standardmäßigen „new"-Operator. Normalerweise dienen zusätzliche Argumente bei „new" dazu, dem Operator näher zu sagen, wie und wo er Speicher allozieren soll.

Ganz schlaue Programmierer werden sich jetzt fragen, warum ein „new" als ersten Parameter immer die Objektgröße haben muß, obwohl man diese Größe doch auch mit „sizeof(Klasse)" berechnen kann. Dafür gibt es gleich zwei gute Gründe:

(1) Die Operatoren „new“ und „delete“ werden vererbt:

```
#include <stddef.h>

class Basis
{ // ...
  public:
    void *operator new(size_t);
};

class Klasse : public Basis
{ // ...irgendwas
};

Klasse *kp = new Klasse; // Benutzt "Basis::operator new"
```

Im allgemeinen ist eine abgeleitete Klasse größer als ihre Basisklasse, und deshalb benötigt ein „new“-Operator wirklich eine präzise Angabe über die Größe des zu reservierenden Speichers.

ABER: Wenn eine Klasse „C“ einen eigenen „new“-Operator besitzt, wird dieser nicht benutzt, um einen Vektor von „C“ einzurichten (z. B. „new C[50]“). Das klingt ziemlich komisch, aber es gehört zu den Prinzipien von C++, daß eine abgeleitete Klasse alle Eignschaften ihrer Basisklassen erbt und folglich auch deren Speicherverwaltung, oder anders ausgedrückt: Da im obigen Beispiel ein „Klasse“ immer gleichzeitig ein „Basis“ ist, macht es auch Sinn, das „new“ von „Basis“ auch für „Klasse“ zu verwenden - vielleicht funktioniert die Basisklasse ja nicht richtig, wenn sie nicht mit ihrem eigenen „new“ alloziert wurde. Auf dem Amiga kann man sich z. B. Klassen vorstellen, die irgendwelche Grafik- oder Sounddaten repräsentieren und deshalb in Chip-Memory liegen müssen, also zwangsläufig ein eigenes, selbstgeschriebenes „new“ benötigen. Folglich müssen dann auch alle abgeleiteten Klassen im Chip-Ram liegen.

Andererseits ist ein „C[50]“ ein Vektor, aber kein „C“, und erbt damit im allgemeinen auch nicht die Eigenschaften von „C“.

(2) Auch das globale „new“, das für alle möglichen Objekte, die kein eigenes „new“ haben, benutzt wird, kann überladen werden, z. B. so:

```
#include <stddef.h>
#include <stream.h>

void *operator new(size_t S, void *Wo)
{ return Wo;
}
```

```
void main()
{ int i = 26731;

  int *ip = new(&i)int(42);

  cout « i; // Ausgabe: 42
}
```

Dieses trickreiche Programm definiert ein neues globales „new", dem man sagen kann, wo es Speicher „reservieren" kann - oder sagen wir besser: an welcher bereits reservierten Adresse das neue Objekt liegen soll. Im Beispiel wird ein „int"-Objekt eingerichtet, indem die Adresse eines bereits (auf dem Stack) existierenden Objekts übergeben wird. Anschließend zeigt „ip" dann auf „i", und „i" wird während dieses „new" erneut initialisiert.

4.2.2.2.3 Der „delete"-Operator

Kein „new" ohne „delete", und so wird es Sie wohl nicht wundern, daß man auch den „delete"-Operator überladen kann.

Eine „delete"-Funktion kann nur innerhalb einer Klasse deklariert werden, und zwar höchstens ein „delete" pro Klasse (d. h. diese Funktion kann nicht überladen werden). Der erste Parameter muß ein „void*" sein und zeigt auf das zu löschende Element, ein optionaler zweiter Parameter vom Typ „size_t" enthält die Objektgröße. Der Ergebnistyp darf ausschließlich „void" sein, und genau wie „new" ist „delete" immer ein statischer Member und folglich auch niemals virtuell.

Zwei Beispiele:

```
#include <stddef.h>

class C
{ // ...
  public:
    void operator delete(void*);
};

class D
{ // ...
  public:
    void operator delete(void*, size_t);
};
```

Auch „delete" wird vererbt, d. h. wenn eine Basisklasse ein überladenes „delete" besitzt, wird dieses auch für die abgeleitete Klasse benutzt. Dabei gibt es aber (scheinbar) ein Problem, das am folgenden Beispiel hoffentlich deutlich wird:

```
#include <stddef.h>
#include <stream.h>

class Basis
{ int i;
  public: // "delete"-Operator:
    void operator delete(void*, size_t);

          // virtueller Destruktor:
    virtual ~Basis(); // Wichtig - siehe unten
};

void Basis::operator delete(void *Obj, size_t Size)
{ cout «  "Das Objekt " « Obj « " der Größe " « Size «
          " wird gelöscht.\n";
  ::delete Obj // Globales "delete" benutzen
}

Basis::~Basis()
{ cout « "Basis-Objekt " « this « " wird destruiert.\n";
}

class Dummy // Dient nur dazu, der anderen
{ int dumm;
};          // Basisklasse einen Offset zu verpassen.

class Klasse : private Dummy, public Basis
{ int i, j, k;
  public:
    ~Klasse();
};

Klasse::~Klasse()
{ cout « "Klassen-Objekt " « this « " wird destruiert.\n";
}
```

```
void main()
{ Klasse *kp = new Klasse;
  Basis  *bp = kp;

  cout « "Adresse des Objekts: " « kp « "\n";
  cout « "Objektgröße: " « sizeof(Klasse) « "\n";
  cout « "Adresse der Basisklasse: " « bp « "\n";
  cout « "Basisklassengröße:   " « sizeof(Basis) « "\n\n";

  // Jetzt wird's spannend:
  delete bp;
}
```

Wir haben hier also eine Klasse „Klasse" mit zwei Basisklassen. Aus der zweiten, „Basis", erbt „Klasse" einen „delete"-Operator. Naturgemäß ist die Basisklasse kleiner als die abgeleitete Klasse und hat auch, da sie in der Basisklassenliste an zweiter Stelle steht, einen Offset der Größe „sizeof(Dummy)".

Das Programm erzeugt also mit dem normalen „new" ein Element der Klasse „Klasse" und weist den Zeiger an eine geeignete Variable zu. Außerdem wird ein Zeiber „bp" auf die Basisklasse gesetzt. Wie Sie sicher wissen, managet der C++-Compiler diese Typkonvertierung korrekt und addiert dabei den richtigen Offset, so daß die numerischen Werte von „kp" und „bp" nicht übereinstimmen.

Nun sagt das Programm, was es getan hat, und gibt Adresse und Größe sowohl des gesamten Objekts als auch der darin enthaltenen Basisklasse aus. Schließlich löscht es das Objekt - aber über den Basiszeiger „bp" und das selbstdefinierte „delete"!

Da der Destruktor von „Basis" als virtuell definiert ist, werden vor dem eigentlichen „delete"-Aufruf auch völlig korrekt erst die abgeleitete Klasse und dann die zweite Basisklasse destruiert (die erste „Dummy"-Basisklasse hat ja keinen Destruktor). So, und jetzt kommt's: „Basis::operator delete" wird aufgerufen, und das ist der Punkt, auf den ich hinaus will und weshalb ich dieses ganze Beispiel überhaupt vorexerziere:

In C++ gibt es keinen „self"-Pointer, d. h. wenn man einen Zeiger auf ein Objekt hat (das möglicherweise in Wirklichkeit nur ein Basisobjekt eines größeren Objekts ist), gibt es keine Möglichkeit, festzustellen, von welchem Typ das referierte Objekt wirklich ist und an welcher Speicheradresse es wirklich beginnt. Der Compiler hat bei „delete bp" also nur einen Zeiger auf die Basisklasse und müßte dem „delete"-Operator nun eigentlich diesen Zeiger auf die Basisklasse sowie deren Größe übergeben, und das wäre hier offensichtlich falsch.

Aber wie so oft in scheinbar ausweglosen Situationen bietet C++ eine elegante Lösung auch für dieses Problem: Wenn die Klasse „Basis" einen virtuellen Destruktor besitzt (so wie es oben der Fall ist), wird dort in der Tabelle der virtuellen Funktionen nicht nur ein

Verweis auf den richtigen Destruktor, sondern auch eine Information über die wahre Objektgröße und den Offset der jeweiligen Basisklasse abgelegt. Im Beispiel stehen dort also der Offset („-sizeof(Dummy)"), der von „bp" zu subtrahieren ist, und die tatsächliche Objektgröße („sizeof(Klasse)"). Also gibt das Programm folgendes aus (wobei die Adressen natürlich andere Werte haben werden, wenn Sie das auf Ihrem Rechner ausprobieren):

```
Adresse des Objekts:      0x2b8d20
Objektgröße:              24
Adresse der Basisklasse: 0x2b8d24
Basisklassengröße:        8

Klassen-Objekt  0x2b8d20  wird destruiert.
Basis-Objekt    0x2b8d24  wird destruiert.
Das Objekt      0x2b8d20  der Größe 24 wird gelöscht.
```

Zum Vergleich: Ohne das kleine Wörtchen „virtual" wird folgendes ausgegeben:

```
Adresse des Objekts:      0x32a878
Objektgröße:              20
Adresse der Basisklasse: 0x32a87c
Basisklassengröße:        4

Basis-Objekt    0x32a87c  wird destruiert.
Das Objekt      0x32a87c  der Größe 4 wird gelöscht.
```

Das Gesamt-Objekt ist hier nur 20 statt 24 Bytes groß, weil der Zeiger auf die Virtual-Tabelle nun entfällt. Diese Speicherersparnis bezahlen wir aber damit, daß der Compiler beim „delete bp" nicht mehr weiß, was eigentlich los ist: Hier wird nur noch der Destruktor für „Basis" aufgerufen, und der „delete"-Operator erhält als Argumente lediglich einen Zeiger auf ein „Basis"-Objekt sowie dessen Größe. Unter Umständen kann die selbstdefinierte Speicherverwaltung damit ganz schön Probleme kriegen, denn ein solches Objekt wurde ja nie zuvor wirklich eingerichtet. Die Heap-Verwaltung von MaxonC++ ist aber robust genug um zu wissen, daß dieses Objekt Basis eines anderen ist.

4.2.2.3 Funktionsaufruf als Operator

Für Leute, die Pascal oder andere „harmlose" Programmiersprachen gewohnt sind, ist es ein gewöhnungsbedürftiger Gedanke, daß der Funktionsaufruf in C++ formal ein Operator ist, dessen linker Operand entweder ein (eventuell überladener) Funktionsname oder ein Ausdruck, der eine Funktion (in einigen Implementationen alternativ auch einen Zeiger darauf, siehe 2.3.2) darstellt, ist, während die Argumentliste als rechter Operand gilt.

Operanden sind, das haben Sie sicher schon gemerkt, zum Überladen da, und folglich kann auch der Funktionaufrufs-Operator, der durch „()“ dargestellt wird, auf Klassen (also ausschließlich als Member-Funktion) überladen werden, z. B. so:

```
class Funny
{ int i;
  public:
    void operator()();
    int operator()(int, int);
};

#include <stream.h>

void Funny::operator()()
{ cout « "Hello, World!\n" « "i = " « i « "\n";
}

int Funny::operator()(int j, int k)
{ i = j+k;
  return i;
}

void main()
{ Funny f;
  int r = f(17,4);
  f();
}
```

Zunächst einmal sollten Sie sich nicht von den diversen Klammern, die sich hier anhäufen, irritieren lassen: erstens ist LISP viel schlimmer und zweitens ist „()“ die Darstellung des Operators, „operator()“ folglich ein Funktionsname und das Klammerpaar, das jeweils folgt, die Parameterliste - also genau wie bei jedem anderen Operator auch. Ebenso leicht ist die Funktion dieser Überladung zu erkennen: Wird ein Klassenobjekt mit einer Argumentliste wie eine Funktion aufgerufen, so ruft der Compiler die entsprechende Operatorfunktion wie eine ganz normale Memberfunktion auf.

Das folgende Programm ist deshalb zum obigen äquivalent, kommt aber ohne Überladen von „()“ aus:

```
class LessFunny
{ int i;
  public:
    void operator_fun();
    int operator_fun(int, int);
};
```

```
#include <stream.h>

void LessFunny::operator_fun()
{ cout « "Hello, World!\n" « "i = " « i « "\n";
}

int LessFunny::operator_fun(int j, int k)
{ i = j+k;
  return i;
}

void main()
{ LessFunny f;
  int r = f.operator_fun(17,4);
  f.operator_fun();
}
```

Wie Sie sehen, dient das Überladen von „()" eigentlich nur dazu, daß man eine Memberfunktion aufrufen kann, ohne ihren Namen nennen zu müssen. Das impliziert auch schon die wichtigste Anwendung dieses Features: Den Operator „()" überlädt man auf Klassen, die in irgendeiner Weise Funktionen repräsentieren. Die Möglichkeit, „()" zu überladen, ist also alles andere als das zentrale Element der objekt-orientierten Programmierung, aber hin und wieder ist es ganz praktisch, weil es elegantere Programme ermöglicht.

4.2.2.4 Klassen als Möchtegern-Vektoren

Ähnlich wie der Funktiosnaufruf, ist auch der Indexoperator „[]" überladbar. „operator[]" muß dabei eine nicht-statische Memberfunktion sein.

Bekanntlich betrachtet C (und damit auch C++) die Vektor-Indizierung als binären Operator, und das ist bei überladener Klassen-Indizierung nicht anders. Dieses Rumgesülze soll ganz einfach zum Ausdruck bringen, daß man auch mit Überladen von „[]" keine echten mehrdimensionale Vektoren realisieren kann, weil der zweiter Operand auch hier nur ein einfacher Ausdruck sein darf:

```
class Vektor
{ int *daten;
  unsigned anzahl;
  public:
    int &operator[ ](unsigned);  // Folgende Zeile geht NICHT:
    int &operator[ ](int, int);  // Keine mehrdimensionalen
                                 // Vektoren!
};
```

```
#include <stream.h>

int &Vektor::operator[ ](unsigned index)
{ if (index < anzahl)
    return daten[index];
  else
  { cout « "Murx!\n";
    exit(26731);
  }
}

void main()
{ Vektor v1;
  int i;

  // ...

  v1[2] = v1[i];
}
```

Rein formal entspricht der Ausdruck „a[b]“ dem Funktionsaufruf „a.operator[](b)“. Wegen dieser Semantik wird weder verlangt, daß der zweite Operand, also der „Index“, irgendwie von einem numerischen Typ sein muß, noch will der Compiler als Ergebnistyp dieser Operation unbedingt eine Referenz (d. h. einen L-Wert) haben. Diese Typen treffen die gewohnte Semantik des Vektor-Zugriffs aber am besten, und so wird man beim Überladen von „[]“ wohl stets über einen diskreten Datentypen indizieren und eine Referenz auf ein Objekt zurückgeben.

Den Operator „[ ]“ wird man naheliegenderweise nur dann überladen, wenn eine Klasse irgendwie so etwas wie einen Vektor repräsentiert. Typische Beispiele sind Strings oder dynamische Vektoren, die sich automatisch dem benötigten Indexbereich anpassen. Man könnte „[]“ aber z. B. auch auf der Telefonlisten-Struktur aus unserem kanonischen Beispielprogramm überladen, und zwar möglicherweise gleich mehrfach:

```
{ class Telefonbucheintrag
  public: char Name[40], Nummer[20];
};

class Telefonbuch  // Irgendwelche Basisklassen...
{ // usw.
  public:
    Telefonbucheintrag &operator[ ] (int nr);
    Telefonbucheintrag &operator[ ] (const char *name);
};
```

```
Telefonbuch T1, T2;

#include <stream.h>

void CrunchyFrog()
{ cout << T1[42].Name << T2["Monty Python"].Nummer;
}
```

In diesem Fall könnte die konventionelle Indizierung mit „int nr" eine Referenz auf das „nr"-te Listenelement liefern, während man bei der Indizierung mit einem String den Telefonbucheintrag mit diesem Namen heraussuchen könnte, was dann einem assoziativen Array entspräche.

Zeiger und Vektoren sind in C eng miteinander verwandt, und deshalb schließt sich das folgende Feature geradezu wunderbar an:

4.2.2.5 Intelligente Zeiger: "->" und das unäre ""*

Beim unären „*"-Operator ist zum Thema Überladung eigentlich nicht viel zu sagen, denn das geht genau wie bei „+" oder „!":

```
// entweder als Member...
class C1
{ C1 operator * ();
};

// oder einfach so:
class C2 { };

int operator *(C2);
```

Witziger ist da schon „->", denn das ist eigentlich kein echter Operator: Normalerweise ist der linke Operand ein Zeiger und der rechte der Name eines Members, also alles andere als ein Audruck. Deshalb weicht die Semantik bei der Überladung von „->" etwas von den „normalen" Operatoren ab.

„->" kann nur als nicht-statische Memberfunktion deklariert werden und hat originellerweise auch keine weiteren Parameter, weshalb weiteres Überladen innerhalb einer Klasse hier flach fällt. Ein Ausdruck wie „x->y" wird, wenn „x" ein Klassenausdruck ist, als

```
(x.operator->())->y
```

ausgewertet. Folglich sollte der Ergebnistyp dieses Operators tunlichst etwas sein, worauf der Compiler dann „->y" auch anwenden kann, also normalerweise ein Zeiger

auf eine Klasse, in der „y" Member ist (alternativ könnte das Ergebnis natürlich auch eine weitere Klasse sein, auf der „->" überladen ist... - aber so schlimm muß man es ja nicht unbedingt treiben). Ein simples Beispiel:

```
struct Objekt   { int a, b, c; };

class Zeiger
{ int foo, bar;
  public:
    Objekt *operator ->();
    Objekt &operator *();
};

Objekt Foobar;

Objekt *Zeiger::operator->()
{ return &Foobar;
}

Objekt &Zeiger::operator*()
{ return Foobar;
}

void main()
{ Zeiger z1, z2;

  // Alle folgenden Operationen manipulieren das Objekt "Foobar":
  z1->a = 42;
  z2->b = 76;
  (*z2).c = 26731;
}
```

Zugegebenermaßen ist es nicht unbedingt sinnvoll, wenn „*" und „->" immer Verweise auf ein einziges Objekt liefern. Typische Anwendungen für solche Überladungen sind „intelligente Zeiger": Beispielsweise könnte man die gerade nicht benötigten Teile einer großen Datenstruktur auf Festplatte oder Diskette auslagern und Objekte möglichst nur dann ins RAM laden, wenn sie benutzt werden. Die Operatoren „*" und „->" müßten dann feststellen, ob das referierte Objekt sich gerade im Speicher befindet, und es andernfalls vom Massenspeicher laden.

4.2.2.6 Inkrement und Dekrement

Die Operatoren „++" und „--" sind außerordentlich bemerkenswert, weil sie als einzige unäre Operatoren gleichermaßen als Postfix- und als Präfixoperatoren benutzt werden können. In ihrer Standardbedeutung haben sie dabei sogar unterschiedliche Funktionen,

denn „++c" und „c++" liefern bekanntlich unterschiedliche Ergebnisse. Das Problem ist nun, was der Compiler machen soll, wenn „++" oder „--" vom Anwender überladen werden.

Im ersten C++-Standard konnte man nur jeweils einen Operatoren definieren, der dann sowohl bei Postfix- als auch bei Präfixausdrücken aufgerufen wurde. Es gab für die Operatorfunktion keine Möglichkeit festzustellen, auf welche der beiden Arten sie benutzt wurde. Es stellte sich aber bald heraus, daß diese Konvention nicht zu befriedigen vermochte, denn schließlich dient der ganze Schmonz mit den überladenen Operatoren hauptsächlich dazu, die gewohnten Funktionsweisen von Operatoren auf selbstdefinierten Datentypen zu imitieren. Also griff man bei C++ 2.0 zu einer anderen Lösung.

Werden die Operatoren „++" und „--" unär überladen, so bezieht sich das auf die Präfix-Schreibweise „++x". Man kann diese Operatoren aber auch mit einem zweiten „Phantom-Operanden" des Typs „int" versehen, und das entspricht dann einem Postfix-Ausdruck:

```
class N
{ int n;
  public:
    N operator ++();
    N operator ++(int);
};

N n1;

void main()
{ ++n1;    // Entspricht n1.operator++()

  n1++;    // Entspricht n1.operator++(0)
}
```

Dem Phantom-Parameter wird also pauschal das Argument „0" übergeben, denn er dient ja ausschließlich dem Zweck, daß die beiden Funktionen sich unterscheiden sollen.

Bei global definierten Funktionen geht das völlig analog:

```
class M
{ public:  int m;
};

M operator ++(M);
M operator ++(M, int);

M m1;
```

```
void main()
{ ++m1;    // entspricht operator++(m1);

  m1++;    // entspricht operator++(m1,0);
}
```

Es gibt eine furchtbar schlaue Begründung dafür, weshalb man das gerade so und nicht genau andersrum definiert hat, aber die ist mir gerade entfallen, und ich möchte Ihnen auch nicht zumuten, sich das zu merken. Im Zweifelsfall wissen Sie ja, wo Sie es nachschlagen können.

Ach ja, auch wenn hier nur Beispiele mit „++" stehen, funktioniert das Überladen von „--" natürlich ganz genauso.

4.2.2.7 Ein- und Ausgabe

So manche Daten, die irgendwie so anfallen, möchte man gern auch ausgeben. Am schönsten wäre es natürlich, wenn das ganz normal ginge, so mit „cout" und so...

Aber hallo, das geht, und wie! „cout" ist nichts anderes als ein bestimmtes Objekt der Klasse „ostream", und der Operator „«", der sonst eigentlich Bits durch die Gegend schiebt, ist mit dieser Klasse als linken Operanden mehrfach überladen. Klar, daß der mündige Programmierer das auch selbst kann.

Also, nehmen wir einmal an, Sie haben eine Klasse namens „Complex", die rein zufällig auch komplexe Zahlen repräsentiert. Nun wäre es natürlich ganz nett, wenn man diese Daten ganz einfach nach „cout" schicken könnte. Das ist nach all dem, was Sie bisher wissen, wirklich kein Thema mehr:

```
// Erster Versuch!

#include <stream.h>  // da wird "ostream" definiert

class Complex
{ public:
    double re, im;
};

void operator « (ostream os, const Complex c)
{ os « c.re « "+" « c.im «"i";
}
```

Das Problem dabei: Wenn „«" ein „void" zurückgibt, wie kann man denn dann mehrere Ausdrücke hintereinander ausgeben, so wie bei „cout « a « b"? Also muß eine derartige

Funktion auf jeden Fall ein Objekt der Klasse „ostream" zurückgeben. Da man aber nie so genau wissen kann, wie ein Objekt reagiert, wenn es kopiert wird, arbeitet man hier besser mit Referenzen, und zwar sowohl beim Funktionsergebnis als auch beim „ostream"-Parameter:

```
// So ist's besser:
#include <stream.h>

class Komplex
{ public:
    double re, im;
};

ostream &operator « (ostream &os, const Komplex &c)
{ os « c.re « "+" « c.im «"i";
  return os;
}

void main()
{ Komplex k;
  k.re = 1.0;
  k.im = 2.0;

  cout « k;
}
```

Da wir gerade so schön mit Referenzen hantieren, deklarieren wir im obigen Beispiel auch den „Komplex"-Parameter als Referenz, immerhin spart eine Referenz im Vergleich zu einem Class-Parameter normalerweise so einiges an Rechenzeit.

Wenn Sie wissen, daß „cin" ein Objekt der Klasse „istream" ist, dürfte Ihnen jetzt auch klar sein, wie man die Eingabe eines Objekts bewerkstelligt:

```
#include <stream.h>

class X
{ public:
    char x[50];
};

istream &operator » (istream &im, X &y)
{ im » y.x;
  return im;
}
```

```
void main()
{ X x1, x2;
  cin » x1 » x2;
}
```

Sie dürfen nur die „return"-Anweisung in solchen Operatorfunktionen niemals vergessen, denn sonst gibt es echt eklige und schwer zu findende Abstürze, aber holla!

4.2.3 Beispielprogramm: Die „string"-Bibliothek

Es dürfte allgemein bekannt sein, daß BASIC die leistungsfähigste und komfortabelste Programmiersprache ist.

Lacht da etwa jemand?

Na gut, leicht übertrieben ist das schon, aber eines muß man BASIC lassen: es gibt keine andere gebräuchliche Programmiersprache, die ein so müheloses und intuitives Stringkonzept besitzt. Man sagt ganz einfach „A$", und der Interpreter weiß Bescheid und erledigt den Rest. Da kann höchstens noch Rexx mithalten, aber Rexx ist weder „gebräuchlich" noch eine „Programmiersprache".

In jeder „echten" Programmiersprache muß man zumindest bei der Deklaration einer Stringvariablen angeben, wie lang das Ding einmal werden soll, und das Stringhandling von C ist bekanntermaßen völlig greuslich, da faktisch eigentlich gar nicht von einem Stringhandling die Rede sein kann. Also sollte man doch besser auf BASIC umsteigen, oder wie?

Keine Angst, Sie haben das Geld, daß Sie für MaxonC++ hingeblättert haben, nicht umsonst verschleudert. Denn erstens hat MaxonC++ eine String-Bibliothek, die fast so komfortabel wie BASIC ist, und zweitens werde ich eine Mini-Version dieser Bibliothek hier im Quelltext vorstellen, denn daran lassen sich viele Konzepte, wie Copy-Konstruktoren, Überladen von Operatoren, Konvertierungsfunktionen oder temporäre Objekte anschaulich darstellen. Dieses Beispielprogramm soll der krönende Abschluß des Kapitels 4 werden.

Das mit dem „$"-Zeichen hinter dem Variablennamen können wir leider (?) nicht von BASIC abkupfern, denn man wird eine Stringvariable deklarieren müssen wie jede andere auch. Dafür dürfen Variablennamen auch mehr als zwei Zeichen lang sein (wie war das doch gleich im C64-BASIC?) und werden ganz einfach mit dem Datentyp „String" ohne jede Längenangabe deklariert. Außerdem werden unsere Strings lauter Dinge können, die mit den normalen C-Strings - oder sagen wir besser „Zeichenvektoren" - nicht so einfach gehen, nämlich Zuweisungen ohne „strcpy", Vergleiche ohne „strcmp" und Aneinanderhängen ohne „strcat".

Also, machen wir uns ans Werk. Sie können sich sicher schon denken, daß wir hier eine Klasse namens „String" deklarieren werden. Die enthält dann lediglich Informationen über den String, während die eigentliche Zeichenkette in einem dynamisch mit „new" reservierten Speicherbereich liegen wird. Auf diese Zeichenkette, die wie immer mit einem Nullbyte abgeschlossen wird, zeigt der private Member „str". Wir vereinbaren dabei, daß der Wert 0 hier eine gültige Repräsentation für den leeren String darstellen soll. Weil es uns fast nichts kostet und an einigen Stellen nutzen wird, speichern wir auch die Stringlänge ab, und zwar im Member „len". Da diese Daten zu den Eigenheiten der Implementierung gehören, ist es sinnvoll, sie als „private" zu deklarieren.

Also geht die Definition unserer Klasse so los:

```
class String
{ char *str;
  int len;
```

Natürlich wird unsere Klasse diverse Konstruktoren haben. Diese und einige anderen Funktionen werden aber etwas gemeinsam haben: Zu einer gegebenen Zeichenkette (ein „char*", nicht etwa ein „String") muß ein Speicherbereich geeigneter Größe mit „new" eingerichten, der String dorthin kopiert, ein Zeiger darauf in „str" eingetragen und die richtige Stringlänge in „len" vermerkt werden. Also deklarieren wir dafür eine (ebenfalls private) Memberfunktion:

```
void alloc (const char*);
```

mit folgender Definition:

```
void String::alloc (const char *initstr)
{ if (initstr)
  { // Stringlänge feststellen, Speicher einrichten:
    len = strlen(initstr);
    str = new char[len+1];
    if (str)  // Neuen String initialisieren:
      strcpy(str, initstr);
    else      // "new" schlug fehl, also "aussteigen":
      exit(42);
  }
  else        // Initialisierung mit Leerstring:
  { len = 0;
    str = 0;
  }
}
```

Das Gegenstück ist die ebenfalls mehrfach benötigte Memberfunktion

```
void free();
```

die den für die Zeichenfolge reservierten Speicher freigibt und aus Gründen der Konsistenz den „str"-Zeiger wieder auf Null setzt:

```
void String::free()
{ if (str)
  { delete [ ] str;
    str = 0;
  }
}
```

Zur Erinnerung: Das „[ ]" hinter dem „delete" ist nötig, weil der Vektor mit variabler Größe eingerichtet wurde.

Zwei Objekte dürfen sich niemals dieselbe physikalische Zeichenkette, also einen allozierten Speicherbereich „teilen", indem die „str"-Zeiger beider Objekte auf dieselbe Adresse zeigen, denn dann kommt es fast zwangsläufig zu Kollisionen. Deshalb werden wir zur Initialisierung von Strings meist die Funktion „alloc" benutzen, die ja eine eigene Kopie ihres Arguments anlegt. Es gibt aber auch die Situation, daß eine Funktion bereits einen Zeichenvektor dynamisch alloziert und initialisiert hat und genau diese Zeichenkette als „str" in ein Stringobjekt eingetragen werden soll, ohne daß sie noch einmal kopiert werden sollte. Für diesen Zweck führen wir einen Konstruktor ein, der nichts anderes tut, als einen Stringzeiger und eine Länge einzutragen:

```
String (int, char*);
```

Da auch dies eine Low-Level-Operation ist, ist auch dieser Konstruktor privat. Seine Definition ist trivial und sieht folgendermaßen aus:

```
String::String (int len, char *str) : len(len), str(str) { }
```

Da gibt es also wirklich nichts mehr zu erklären - die Datenmember des Objekts werden mit bestimmten Werten initialisiert, und das ist es dann.

Kommen wir nun zu den öffentlichen Konstruktoren: Wir brauchen eigentlich deren drei, nämlich einen Default-Konstruktor, der ein „String"-Objekt mit dem Leerstring initialisiert, einen Konstruktor, der einen gewöhnlichen „char[]"-String in ein Objekt der Klasse „String" verwandelt und somit einen Teil der Schnittstelle zwischen unserer Stringbibliothek und der Standard-Stringverwaltung von C darstellt, und einen Kopier-Konstruktor, denn es dürfen ja niemals zwei String-Objekte auf denselben Zeichenvektor verweisen, weshalb beim Kopieren von Objekten immer auch eine Kopie der Zeichenkette angelegt werden muß.

Mit Hilfe eines Default-Arguments können wir aus diesen drei aber zwei machen:

```
public:  String (const char* = 0);
  String (const String &);
```

Auch deren Definition ist trivial, da wir ja schon die äußerst nützliche Funktion „alloc" haben:

```
String::String (const char *initstr)
{ alloc(initstr);
}

String::String (const String &s)
{ alloc (s.str);
}
```

Nun fehlt uns noch ein Destruktor:

```
~String();
```

Die Funktion „free" macht eigentlich schon alles, was der Destruktor leisten muß:

```
String::~String()
{ free();
}
```

Wir haben einen Konstruktor, der eine Zeichenfolge in einen String umwandelt. Da liegt es doch nahe, eine Konvertierungsfunktion zu schreiben, die gerade das Gegenteil macht. Zur Abwechslung deklarieren wir sie gleich als „inline"-Member innerhalb der Klassendefinition:

```
operator char*() const
{ if (str)
    return str;
  else
    return "";
}
```

Bei einer Wertzuweisung unter Strings müssen wir denselben Aufwand treiben wie beim Kopier-Konstruktor, nämlich eine Kopie des Zeichenvektors anlegen. Also ist auch noch der Operator „=" zu überladen:

```
String &operator = (const String &);
```

Auch diese Memberfunktion hat eine vergleichsweise triviale Definition: Wir geben den alten Stringinhalt frei, tragen einen Zeiger auf den kopierten neuen Wert ein und geben eine Referenz auf das soeben initialisierte Objekt zurück:

```
String &String::operator = (const String &s)
{ free();
  alloc(s.str);
  return *this;
}
```

Fällt Ihnen etwas auf? Hier gibt es eigentlich nur ausgesprochen kurze Funktionen. Das ist so auch voll und ganz beabsichtigt, denn objektorientierte Programmierung soll ja dazu führen, daß die Algorithmen trivial werden, indem man die Datenstrukturen intelligent wählt.

Also geht es ebenso trivial weiter: Wir bieten dem Benutzer der Klasse noch einen schnellen und bequemen Zugriff auf die Stringlänge an, wieder in Form einer „inline"-Memberfunktion:

```
int length() const
{ return len;
}
```

Nun bleibt nur noch eins: Der Operator „+", der zwei Strings aneinander hängen soll, muß auf intime Daten der Klasse zugreifen und wird deshalb als „friend" deklariert:

```
friend String operator + (const String &s1, const String &s2);
```

Da wir auf die Funktionen aus „<string.h>" zurückgreifen können, ist es nicht weiter aufwendig, diese Funktion zu implementieren:

```
String operator + (const String &s1, const String &s2)
{ // Wie lang ist das Ergebnis?
  int newlen = s1.length()+s2.length();

  // Speicher anfordern:
  char *newstr = new char[newlen+1];  // Wenn's geklappt
  // hat,dann String darin montieren:
  if (newstr)
  { strncpy (newstr,s1,s1.length());
    strncat (newstr,s2,s2.length())
  }

  // Mit dem "Spezialkonstruktor" wird aus Länge und Stringzeiger
  // ein Stringobjekt erzeugt:
  return String(newlen, newstr);
}
```

Spaßeshalber implementieren wir jetzt auch noch den Vergleichsoperator „==", der zwei Strings mit „strcmp" auf zeichenweise Gleichheit testet. Diese Funktion muß weder als

Member noch als „friend" deklariert werden, da sie auch so an alle für sie notwendigen Daten herankommt.

```
int operator == (const String &s1, const String &s2)
{ return !strcmp(s1,s2);
}
```

Der Aufruf von „strcmp" geht hier absolut locker und problemlos, da ja eine Konvertierungsfunktion von „String" nach „char*" existiert.

Man beachte, daß diese Funktion nicht die gewöhnliche Vergleichsoperation auf Zeigern und Vektoren überdeckt: Bei

```
char *s;    if (s == "Nanu") ...
```

werden jetzt immer noch die Adressen der beiden Zeichenketten verglichen. Unser selbstdefinierter Operator wird also nur aufgerufen, wenn mindestens ein Operand ein Objekt der Klasse „String" ist:

```
String t;    if (t == "Jau") ...
```

Natürlich kann man das auch durch ein explizites Casten erzwingen:

```
char *u;
if (String(u) == String("Jawoll!")) ...
```

Zu unserem Glück fehlt uns noch eine Möglichkeit, Strings auszugeben. Natürlich kann man das mit

```
String x;
cout « (char*)x;
```

bewerkstelligen, aber erstens kann man hier den Cast wegen der am Ende von 4.1.2.1 genannten Gründe nicht weglassen und zweitens ist diese Methode sehr unsicher, wenn der Compiler ein temporäres Objekt einführt, etwa bei

```
String x, y;
cout « (char*)(x+y);        // Vorsicht!
```

Deshalb überladen wir einfach den Operator „«", so wie es in 4.2.2.7 gelernt haben:

```
ostream &operator « (ostream& out, const String& s)
{ return out « (char*)s;
}
```

Die eigentliche Ausgabe und die Rückgabe der Referenz in einer einzigen Anweisung zu verbinden, ist zwar trickreich, aber natürlich erlaubt, denn „out « xxx" gibt ja generell eine Referenz auf „out" zurück, und genau das soll auch unsere eigene Operatorfunktion als Ergebnis liefern.

Haben Sie die Übersicht verloren? Das kann ich gut verstehen, denn in den Listing-Fragmenten ging es oben doch manchmal etwas durcheinander. Deshalb kommt jetzt noch einmal das Ganze Programm im Überblick:

```
   // Stringbibliothek (verkleinerte Version von „tools/str.h")
   // *** Deklaration der Klassen und Funktionen:

class String
{ char *str;
  nt len;
  void alloc (const char*);
  void free();
  String (int, char*);
  public: String (const char* = 0);
  String (const String &);
  ~String();
  operator char*() const
  { if (str)
      return str;
    else
      return "";
  }
  String &operator = (const String &);
  int length() const
  { return len;
  }
  friend String operator + (const String &s1, const String &s2);
};

int operator == (const String &s1, const String &s2);

class ostream& operator « (ostream&, const String&);

// *** Implementierung:

#include <string.h>
#include <stream.h>
```

```
void String::alloc (const char *initstr)
{ if (initstr)
  { len = strlen(initstr);
    str = new char[len+1];
    if (str)
      strcpy(str, initstr);
    else
      exit(42);
  }
  else
  { len = 0;
    str = 0;
  }
}

void String::free()
{ if (str)
  { delete [ ] str;
    str = 0;
  }
}

String::String (int len, char *str) : len(len), str(str) {
}

String::String (const char *initstr)
{ alloc(initstr);
}

String::String (const String &s)
{ alloc (s.str);
}

String::~String()
{ free();
}

String operator + (const String &s1, const String &s2)
{ int newlen = s1.length()+s2.length();
  char *newstr = new char[newlen+1];
  if (newstr)
  { strncpy (newstr,s1,s1.length());
    strncat (newstr,s2,s2.length())
  }
  return String(newlen, newstr);
}
```

```
String &String::operator = (const String &s)
{ free();
  alloc(s.str);
  return *this;
}

int operator == (const String &s1, const String &s2)
{ return !strcmp(s1,s2);
}

ostream &operator « (ostream& out, const String& s)
{ return out « (char*)s;
}

// *** Test- und Beispielprogramm:

void main()
{ String s1, s2="I didn't expect the", s3;
  s1 = "Spanish Inquisition";
  s3 = s2 + " " + s1;

  char *awurx = "Vor"+"sicht"; // Unsichere Operation, da
                               // temporäres Objekt!
  cout « s3+".\n";
}
```

5. Der Preprozessor

5.1 Allgemeines

Ich weiß, es ist für Sie ebenso wie für mich eine Zumutung, aber trotzdem muß ich mich jetzt nach den ganzen tollen objektorientierten Features über etwas so primitives wie den Preprozessor auslassen. Wie so ziemlich alles in C, kann man mit dem Preprozessor furchtbar starke Sachen machen, aber sein Programm auch dermaßen verwirren, daß es niemand mehr versteht.

Theoretisch ist der Preprozessor ein eigenständiges Programm, das den Quelltext vorverarbeitet, bevor der C- oder C++-Compiler ihn zu sehen bekommt. Das Ergebnis ist im Prinzip ein vereinfachter Quelltext, bei dem der Compiler sich nicht mehr um triviale Dinge zu kümmern braucht, insbesondere um das Einbinden von Includes, Definieren und Ersetzen von Macros, Weglassen vom Kommentaren und bedingte Übersetzung, also das Einfügen bzw. Weglassen von Quelltextteilen in Abhängigkeit von Bedingungen.

Preprozessor und Compiler arbeiten normalerweise (aber was ist in Zeiten wie diesen schon normal?) völlig unabhängig voneinander, und keinen interessiert, was der andere macht oder gemacht hat. Deshalb könnte man einen C-Preprozessor durchaus für die Vorverarbeitung von Pascal-Programmen oder irgendwelchen anderen Texten benutzen, und es soll tatsächlich Leute geben, die so etwas tun.

Der Preprozessor von MaxonC++ ist fest in den Compiler integriert. Dadurch ist es nicht möglich, einen Text nur durch den Preprozessor laufen zu lassen, ohne ihn zu compilieren. Der Vorteil dieses Verfahrens ist, daß der Compiler dadurch schneller wird und nirgendwo einen zusätzlichen Zwischencode ablegen muß. Wenn Sie andererseits wirklich das Bedürfnis verspüren sollten, einen Text vorverarbeiten zu lassen, ohne ihn durch den Compiler jagen zu wollen, muß ich Sie auf die PD-Serie verweisen. Der MaxonC++ Preprozessor ist integrer Bestandteil des MaxonC++ Compiler Systems, und daran ist nichts zu ändern.

Gesteuert wird der Preprozessor über Quelltextzeilen, die mit einem „#“ anfangen. Dahinter hat normalerweise der Name eines Proprozessorbefehls zu stehen. Unser kleiner Freund befolgt dann diesen Befehl und löscht ihn dann natürlich rückstandlos aus dem Quelltext, so daß der Compiler normalerweise nichts davon mitbekommt. Zwei solcher Anweisungen wurden in diesem Handbuch bereits benutzt: „#include“ zum Einbinden von Headerdateien mit Definitionen und „#pragma“ zum Umschalten zwischen ANSI C- und C++-Modus (wobei „pragma“ noch wesentlich mehr kann).

Jedenfalls muß das „#“ immer an Zeilenanfang stehen, eventuell mit ein paar Leerzeichen davor, und die Preprozessoranweisung geht genau bis zum Zeilenende. Hier liegt der Preprozessor „schief“ zum Compiler, denn in C und C++ ist es ja vollkommen belanglos, wo eine Zeile anfängt oder aufhört. Wesentlich konsistenter wäre es, wenn man Preprozessoranweisungen an beliebiger Stelle mit einem „#“ beginnen und dann mit einem „;“ oder einem anderen geeigneten Zeichen abschließen könnte, aber man hat den Sprachstandard nun einmal anders definiert und mich fragt ja keiner. Merken Sie sich also, daß der Preprozessor streng zeilenorientiert arbeitet und ich ihn deshalb nicht besonders mag.

Aber nun zu „#define“, einer der nützlichsten Preprozessorfunktionen überhaupt.

5.2 Was Sie schon immer über #define wissen wollten, aber nicht zu fragen wagten

5.2.1 Makrodefinition

Wie jeder alte Grieche weiß, heißt „makros“ (mit Betonung auf der letzten Silbe) so viel wie „groß“. Fragen Sie mich aber bitte nicht, was das mit den Makros zu tun hat, die Gegenstand dieses Abschnitts sind.

Nach einer Zeile wie

```
#define Name Weiss der Geier was!
```

ersetzt der Preprozessor überall im Quelltext den Bezeichner „Name“ durch „Weiss der Geier was!“. Hinter dem „#define“ hat als erstes der Makroname zu stehen, der stets ein Bezeichner ist.

Offensichtlich kann man dieses Feature benutzen, um häufig wiederkehrende Texte abzukürzen, aber auch, um das Aussehen des Quelltexts stark zu verändern - dazu später mehr. Im frühen C-Standard war dies außerdem die einzige Möglichkeit, Konstanten zu definieren, etwa so:

```
#define N 100

int Vektor[N];
```

Nach der „define“-Anweisung wird das Makro „N“ überall durch die Zahl „100“ ersetzt. Unschön ist dabei, daß hier keine Scoperegeln gelten, denn der Preprozessor hat bekanntlich nicht so ganz viel mit dem Compiler zu tun:

```
void f()
{
#define x 50
}

void g()
{
#define x "Hello, World!" // ERROR: "x" re-defineirt
}
```

Es gibt für solche Makronamen nur einen einzigen, globalen Gültigkeitsbereich. Ein Makro darf allerdings zweimal mit dem gleichen Wert definiert werden, wobei eine unterschiedliche Anzahl von Leerzeichen egal ist:

```
#define SUM a+b
#define SUM a + b
```

Außerdem kann man eine Makrodefinition löschen, und zwar mit der Preprozessor-Anweisung „#undef“:

```
#define GANZ int

 GANZ i1, i2;   // OK

#undef GANZ

GANZ i3, i4;    // ERROR
```

Ein „Token“ ist ein einzelnes Quelltextsymbol, also ein Bezeichner (z. B. „xyz“), eine Konstante (numerisch oder String), ein reservieres Wort (wie „int“ oder „private“), ein Operator (z. B. „+“) oder ein sonstiges Trennzeichen („[„, „;“). Dagegen sind Kommentare, Leerzeichen oder Preprozessoranweisungen keine Token. Der Preprozessor wandelt den Quelltext in eine Folge von Token um, die der Compiler dann direkt verarbeiten kann. Die Ersetzung von Makros erfolgt in C streng tokenorientiert:

```
#define PLUS +

int i;

void f()
{ i PLUS= 1;
} // ERROR
```

ist ein Fehler, denn obwohl „PLUS“ und „=“ direkt aufeinander folgen, ergeben sich daraus nach der Makroexpansion die beiden Token „+“ und „=“ und nicht etwa das Token „+=“.

Ich kann nicht oft genug wiederholen, daß der Preprozessor keine Ahnung von C hat und der Compiler nicht weiß, wie die Tokenfolge seiner Eingabe zustande gekommen ist. Also werden Bindungsregeln und der Vorrang von Operatoren total ignoriert, womit man ganz schön heftig auf die Schnauze fallen kann:

```
#include <stream.h>

#define N 17
#define M N+4

void main()
{ cout « 2*M;
}
```

Die Ausgabe ist hier „38" und nicht etwa „42", denn da der Preprozessor gnadenlos Token ersetzt, wird „2*M" als „2*17+4" expandiert. Deshalb sollte man Makros, in denen Operatoren vorkommen, immer klammern:

```
#define M (N+4)
```

Stillschweigend habe ich Ihnen hier ein verschachteltes Makro untergejubelt. In der Tat darf man Makros benutzen, um andere zu definieren, so wie oben „N" in der Definition von „M" vorkommt. Dabei werden bei der Definition eines Makros darin enthaltene Makros nicht sofort, sondern erst bei der Benutzung des Makros expandiert:

```
#define N 100
#define M (2*N)
#undef N

int i = M;    // ERROR: Bezeichner "N" unbekannt
```

Da „N" undefiniert wird, ist es im folgenden ein ganz gewöhnlicher Bezeichner und wird vom Preprozessor auch als solcher an den Compiler weitergegeben, d. h. „M" wird hier wirklich als „(2*N)" und nicht als „(2*100)" expandiert.

Übrigens arbeitet der Preprozessor in mehreren Schritten, wobei das Entfernen vom Kommentaren logisch gesehen lange vor dem Ersetzen und Definieren von Makros kommt. Deshalb wird folgendes Makro nicht das Gewünschte leisten:

```
#define REM //

REM Dies soll ein Kommentar wie in BASIC sein,
REM ist aber leider falsch!
```

Hier werden zuallererst Kommentare, also auch der leere Komentar „//“, aus dem Quelltext gekickt, und erst dann stellt der Preprozessor fest, daß „REM“ als Makro definiert wird - und zwar als leeres Makro. Also ersetzt er im nachfolgenden Quelltext „REM“ wunschgemäß überall durch nichts, aber das, was jeweils danach in der Zeile steht, bleibt unverändert stehen und wird an den Compiler weitergegeben.

Es gibt übrigens eine Art Konvention, daß Makronamen immer aus Großbuchstaben bestehen sollten. Dadurch soll es etwas leichter werden, in einem Quelltext zu erkennen, was nun „echt“ ist und was noch vom Preprozessor verändert werden wird.

Makros sind ein wunderbares Mittel, seinen Quelltext vollkommen undurchschaubar zu machen. Auf dem UseNet kursieren schon Tips für alle, die einen Programmierer ärgern wollen: Fügen Sie doch einfach bei einem „Freund“ in eines seiner Includefiles, z. B. in „<stream.h>“, ein paar neue „Makros“ ein, etwa so:

```
#define while if
#define double int
```

Das bedauernswerte Opfer dürfte ziemlich lange brauchen, um herauszufinden, warum ein Programm nicht so läuft, wie es soll...

5.2.2 Makros mit Parametern

Erheblich cooler sind allerdings Makros mit Parametern. Man deklariert sie, indem man unmittelbar hinter den Makronamen eine geklammerte Bezeichnerliste setzt, zum Beispiel wie im folgenden beispielhaften Beispiel:

```
#include <stream.h>

#define ABS(n) ((n)>0 ? +(n):-(n))
#define MAX(p1,p2) ((p1)>(p2) ? (p1) : (p2))

void main()
{ int i = ABS(-7), j = MAX(i+1,9);
  cout « i « " " « j « endl;
}
```

Beim Makroaufruf ist dann für jeden Parameter eine beliebige Tokenfolge anzugeben. Die Argumente sind durch Kommata zu trennen, wobei Kommas innerhalb von Klammern, wie das erste Komma im folgenden Beischlaf:

```
MAX(f(1,2), 42);
```

nicht als Trennung gelten, so daß obiger Makroaufruf korrekt ist.

Die einzige Anforderung an die Makro-Argumente ist, daß ihre Anzahl stimmen muß. Davon einmal abgesehen kann es sich um beliebige Tokenfolgen handeln, und als solche werden sie vom Preprozzessor auch behandelt. Deshalb ist es meist empfehlenswert, die Parameternamen in der Makrodefinition zu klammern:

```
#define DIFF(a,b) a-b

int i = DIFF(42, 17+4);
```

führt zu einem wahrscheinlich unerwarteten Ergebnis, denn der Aufruf von „DIFF" wird hier als „42-17+4" expandiert. Klüger ist deshalb

```
#define DIFF(a,b) (a)-(b)
```

Wenn man aber in der Makrodefinition ein „#" vor einen Parameternamen stellt, wird das zugehörige Argument als String-Literal expandiert:

```
#define TEST(string) cout « "Hello " « #string « "!";

#include <stream.h>

void main()
{ TEST(World)
}
```

Wird der Makroaufruf zu

```
cout « "Hello " « "World" « "!";
```

expandiert, d. h. der Preprozessor setzt gewissenmaßen Anführungszeichen um den Parameter.

Dagegen dient der Operator „##" zum Verbinden von Token:

```
#define MYIDENT(x) geheim_ ## x

int MYIDENT(i);
```

Durch das „##" wird der Makroaufruf nicht mit „geheim_ i", sondern dem einzelnen Token „geheim_i" ersetzt. Dabei muß das Ergebnis der Verbindung aber immer ein einzelnes Token sein:

```
MYIDENT(!)
```

ist ein falscher Aufruf, denn „geheim_" und „!" bilden zusammen kein gültiges Token.

Makros mit Argumenten sind sehr beliebt, um solche Mini-Funktionen wie „ABS" oder „MAX" zu implementieren. Als C++-Programmierer sollte man sich aber überlegen, ob man für so etwas nicht doch lieber eine „inline"-Funktion benutzt, denn wenn der Compiler alle möglichen Optimierungen anwendet, ist das genauso effektiv.

5.3 Was Sie noch nie über #include wissen wollten, aber jetzt trotzdem erfahren

Die Preprozessoranweisung „#include" fügt den Inhalt einer Datei an der entsprechenden Stelle in den Quelltext ein. Als einziges Argument hat „#include" einen Dateinamen, wobei es zwei verschiedene Möglichkeiten gibt, diesen Dateinamen zu schreiben:

```
#include <name>
```

mit den spitzen Klammern sucht die Datei zuerst in einem ganz bestimmten Verzeichnis, wobei es verschiedene Möglichkeiten gibt, dieses Verzeichnis festzulegen:

- In der integrierten Entwicklungsumgebung von MaxonC++ legt man das Verzeichnis mit dem Menüpunkt „Optionen/Suchpfade" fest, oder man stellt den gewünschten Pfad mit dem „CPrefs"-Programm ein.

- In der Kommandozeilenversion ist der Pfad „C++:include" als Default angegeben, den man mit der Option „-i" beliebig anders setzen kann. Damit kann man auch mehrere Pfade angeben, die der Preprozessor dann nacheinander nach der gewünschten Datei absucht.

Man benutzt diese Form von „#include", um Standard-Includefiles einzubinden, z. B. <stdio.h> oder <stream.h>, aber auch die Amiga-Systemincludes.

Als kleines Bonbon vor allem für Diskettenbenutzer gibt es auch noch die Möglichkeit, die benutzten Dateien auf einen schnelleren Datenträger umkopieren zu lassen: Wenn Sie einen doppelten Suchpfad angeben, sucht der Preprozessor zuerst unter dem ersten Pfad, der z. B. in der RAM-Disk liegen könnte. Findet er die Datei dort nicht, versucht er es über den zweiten Pfad, z. B. auf einer Diskette, und kopiert die Datei in das erste Verzeichnis um. Beim nächsten Übersetzungsvorgang liegt die Datei dann auf dem schnellen Medium vor. Übrigens legt der Preprozessor vor dem Umkopieren bei Bedarf selbständig die benötigten Unterverzeichnisse an.

Bei der zweiten Möglichkeit, ein Includefile einzubinden, setzt man den Dateinamen wie einen ganz gewöhnlichen String in Anführungszeichen:

```
#include "name"
```

Nun sucht der Preprozessor die Datei im aktuellen Verzeichnis. Diese Variante dient hauptsächlich dazu, selbstgeschriebene Dateien einzufügen. Der für Standard-Includes vorgesehene Pfad wird dabei zunächst nicht beachtet und folglich die Datei auch nicht umkopiert.

Unter MaxonC++ sind die beiden Varianten von „#include" allerdings praktisch gleichwertig, denn bei der <xxx>-Syntax versucht der Preprozessor es zuerst über den Standard-Pfad und bei Mißerfolg dann auch noch über das aktuelle Verzeichnis, und bei „xxx" ist die Reihenfolge gerade umgekehrt.

5.4 Bedingte Übersetzung

Mit dem Preprozessor kann man zur Übersetzungszeit Entscheidungen treffen, nämlich Teile des Quelltexts aufgrund bestimmter Bedingungen aus dem Quelltext auszublenden. Dazu dienen Konstrukte der Form

```
#if Bedingung

#endif
```

Ist die „Bedingung" erfüllt (was in C ja immer „ungleich Null" heißt), wird der nachfolgende Text gelesen, andernfalls wird alles bis zum „#endif" übersprungen.

Die Bedingung ist hier ein konstanter Ausdruck, der ganzzahlig ausgewertet wird. Dieser Ausdruck kann aus numerischen Konstanten und allen damit sinnvollen Operatoren bestehen. Makros werden im Ausdruck expandiert und alle anderen Bezeichner durch die Konstante „0" ersetzt. Außerdem können Ausdrucke wie

```
defined Name
```

oder

```
defined(Name)
```

darin auftauchen. Das Ergebnis ist „1", wenn „Name" als Makro definiert ist, und andernfalls „0".

Eine Zeile

```
#if defined x
```

kann ersetzt werden durch

```
#ifdef x
```

und entsprechend

```
#if !defined y
```

durch

```
#ifndef y
```

Die Abfrage, ob ein Symbol definiert ist, ist die wohl häufigste Form der bedingten Übersetzung. Beispielsweise geht es bei Include-Dateien oft ziemlich durcheinander, so daß es nicht immer ganz einfach ist, sicherzustellen, daß jede Datei nur einmal gelesen wird. Solche Probleme löst man normalerweise, indem man in jeder Datei ein bestimmtes Symbol definiert, anhand dessen der Preprozessor feststellen kann, ob er diese Datei schon gelesen hat:

```
#ifndef GANZ_SPEZIELLER_NAME
#define GANZ_SPEZIELLER_NAME

// Hier steht der eigentliche Inhalt der Datei.

#endif
```

Die Amiga-Systemincludes machen von diesem Schema ausgiebig Gebrauch.

Fast wie im richtigen Leben gibt es auch zu „#if" ein „#else":

```
#if Bedingung

// Text1

#else

// Text2

#endif
```

Die Bedeutung dieses Konstrukts dürfte klar sein: Abhängig von der Bedingung wird entweder der erste oder der zweite Text gelesen und der jeweils andere übersprungen.

Ein Äquivalent zum „switch" hat der Preprozessor nicht, aber er erleichtert die Konstruktion von Mehrfachabfragen durch „#elif", was sich natürlich von „else if" ableitet

```
#define N 2

#if N == 0    Text #0
#elif N == 1    Text #1
#elif N == 2    Text #2
#else    Text #3
#endif
```

Der „else"-Teil ist bei einer „if"-"elif"-Konstruktion natürlich optional. Gibt es kein „#else" und ist keine der Bedingungen wahr, wird eben das Ganze Konstrukt übersprungen.

Abschließend soll noch darauf hingewiesen werden, daß man solche Abfragen auch ineinander verschachteln kann. Gäbe es z. B. kein „#elif", müßte man für obige Konstruktion folgendes schreiben:

```
#define N 2

#if N == 0   Text #0
#else
#if N == 1    Text #1
#else
#if N == 2     Text #2
#else
Text #3
#endif
#endif
#endif
```

5.5 Pragmatismus

5.5.1 Allgemeines und Wiederholungen

Bei jedem Standard kommt irgendwann der Punkt, wo er nicht mehr ausreicht. Deshalb ist im ANSI-Standard eine Möglichkeit vorgesehen, den Sprachumfang in einer Implementierung zu erweitern, ohne daß es dadurch zu Inkompatiblitäten zwischen den verschiedenen Compilern kommt: Die Preprozessor-Anweisung „#pragma".

Jede Implementierung kann selbst festlegen, was in einer Pragma-Zeile stehen darf und was es bedeuten soll. Es wird nur verlangt, daß eine unbekannte Syntax - also z. B. Pragmas eines anderen Compilers - nicht als Fehler gemeldet, sondern ignoriert werden muß. Das hat leider auch die Folge, daß der Preprozessor versehentlich gemachte Fehler nicht als solche meldet und man sich nachher vielleicht wundert, warum ein #pragma keine Wirkung hat.

Ein paar #pragma-Anweisungen wurden in diesem Handbuch schon an früherer Stelle vorgestellt, aber der Vollständigkeit halber seien sie hier noch einmal vorgestellt:

```
#pragma -
```

schaltet den Compiler in den ANSI C-Modus.

```
#pragma +
```

geht wieder zurück in den C++-Modus.

```
#pragma chip
```

legt alle nachfolgend deklarierten statischen Daten im CHIP-RAM ab.

```
#pragma fast
```

schaltet wieder in den normalen Modus zurück, d. h. alle Daten werden da abgelegt, wo gerade Platz ist.

5.5.2 Breakpoint-Kontrolle

MaxonC++ kann Programme erzeugen, die vom Benutzer abgebrochen werden können. Das ist einerseits während der Programmentwicklung angenehm, wenn man versehentlich eine Endlosschleife programmiert hat, und andererseits ist es oft sinnvoll, wenn der

„normale“ Anwender eines Programms die Programmausführung mehr oder weniger gewaltsam abbrechen kann.

Ein Programm fängt einen Abbruch ab, indem es an bestimmten Stellen, z. B. mindestens einmal innerhalb jeder Schleife, das Signalbit prüft, das durch die Tastenkombination „Ctrl-C“ oder das CLI-Kommando „break“ gesetzt wird. Natürlich kosten diese Tests Rechenzeit, einmal ganz zu schweigen von den dafür nötigen zusätzlichen Anweisungen. Außerdem ist es oft gar nicht wünschenswert, wenn eine bestimmte Aktion einfach so abgebrochen wird - man stelle sich einmal eine Textverarbeitung vor, die bei der Tastenkombination „^C“ einfach aussteigt, ohne vorher abzuspeichern... Lange Rede, schwacher Sinn: Natürlich ist diese Abbruchmöglichkeit optional. Sie können Sie aktivieren, indem Sie die entsprechende Option im Optionen-Dialogfenster bzw. die Compileroption „-gb“ in der Kommandozeilenversion benutzen, andernfalls ist sie deaktiviert.

Aber auch das ist in der Praxis noch nicht der Weisheit letzter Schluß: Vielleicht möchte man ja in einigen Programmteilen die Abbruchmöglichkeit benutzen und in anderen nicht. Dafür gibt es in MaxonC++ das Pragma „break“.

```
#pragma break +
```

schaltet das Setzen der Breakpoints ein und

```
#pragma break -
```

wieder aus.

Ein Beispiel:

```
void main()
{ int i, j=0;

#pragma break-

  for(i=0; i<100000; ++i)
    j = i-j;

#pragma break+

  for(i=0; i<100000; ++i)
    j = i-j;
}
```

In der ersten „for“-Schleife kann das Programm nicht abgebrochen werden, in der zweiten aber schon. Drückt man „^C“, während noch die erste Schleife abgearbeitet

wird, geschieht zunächst einmal nichts. Erst zu Beginn der zweiten Schleife schaut das Programm wieder nach, ob das fragliche Signal eingetroffen ist, und steigt mit der Meldung

```
*** User break
```

aus. Der Abbruch entspricht dabei einem „exit(900)". d. h. es werden Dateien geschlossen und Speicher und sonstige Resourcen freigegeben. Auch mit „atexit" eingehängte Funktionen werden noch ausgeführt. Natürlich ist ein solcher Abbruch ein typisches Low-Level-Konstrukt, so daß dabei keine Destruktoren aufgerufen werden.

Ähnlich wie bei „for" werden auch bei „while"- und „do"-Schleifen sowie bei Schleifenverdächtigen „goto"-Sprüngen solche Abbruchtests eingefügt. Schlauberger werden jetzt natürlich wissen wollen, was bei

```
#pragma break+
while(Bedingung)
{ #pragma break-
}
```

ist, ob also der Wert zählt, der beim Schleifenanfang eingestellt ist, oder der, der beim Schleifenende gilt. Dazu kann ich nur sagen, daß noch nicht einmal ich das auswendig weiß und so etwas sich auch von Version zu Version ändern kann. Wenn man also „pragma break" benutzt, sollte man das so tun, daß es eindeutig ist.

Bleibt noch die Frage offen, was eine Abbruchkontrolle unter MaxonC++ nun wirklich kostet. Sie ist wohl aufs Äußerste optimiert und sieht in Assembler so aus:

```
move.l Signaladresse, An
btst   #Signalnummer, (An)
bne    Ausstieg
```

Diese drei Instruktionen machen sich wirklich nur in sehr engen Schleifen bemerkbar. Die Laufzeit einer leeren „for"-Schleife wird dadurch z. B. noch nicht einmal verdoppelt, und bei Schleifen, die wirklich etwas tun (also in der Praxis so ziemlich alle) ist der relative Zeitverlust entsprechend gering.

5.5.3 Amiga-spezifische Systemaufrufe

In Abschnitt 2.2.1.2.4 wurde dargestellt, wie Routinen des Amiga-Betriebssystems aufgerufen werden, und außerdem wurde Ihnen versprochen, daß es außer der dort beschriebenen Pseudo-Link-Spezifikation noch eine andere Notation gibt, mit der man Systemfunktionen deklarieren kann. Voila, hier ist sie:

Wie Sie sich vielleicht denken können, hat auch dies mit einem „pragma" zu tun, und zwar mit dem pragma „amicall". Eine solche Deklaration besteht im Wesentlichen aus vier Teilen:

1. Der Name der Basisvariablen

2. Der Offset als positive Zahl (d. h. der Compiler ergänzt sich das Minuszeichen selbst).

3. Der Funktionsname. Die Funktion muß bereits deklariert worden sein und darf aus Gründen der Eindeutigkeit nicht überladen sein.

4. Die Parameterliste, vertreten durch eine entsprechende Anzahl von Registernamen in runden Klammern.

Um das Ganze setzen wir ein Paar runder Klammern und noch „#pragma amicall" davor, und fertig ist die Deklaration.

Ein Beispiel:

```
#pragma amicall(DiskfontBase, 30, OpenDiskFont(a0))
#pragma amicall(DiskfontBase, 36, AvailFonts(a0,d0,d1))
#pragma amicall(DiskfontBase, 42, NewFontContents(a0,a1))
#pragma amicall(DiskfontBase, 48, DisposeFontContents(a1))
#pragma amicall(DiskfontBase, 54, NewScaledDiskFont(a0,a1))
```

Auch hier sei noch erwähnt, daß Sie solche Deklarationen normalerweise nicht selbst werden schreiben müssen, denn Sie bekommen sie für alle Amiga-Libraries mitgeliefert.

Nun werden sich sicher einige beschweren, daß MaxonC++ hier zwei unterschiedliche Konzepte für dieselbe Sache hat. Das ist natürlich richtig, aber beide Varianten haben ihre Vorteile:

- Die Link-Spezifikation ist in die Syntax der Sprache selbst integriert, was formal schöner als eine von außen „aufgepfropfte" Preprozessoranweisung ist. Außerdem eignet sich diese Variante auch für überladene Funktionsnamen, was zugegebenermaßen bei Betriebssystemfunktionen eher selten bis gar nicht vorkommt.

- „pragma amicall" ist der de-facto-Standard, für viele Amiga-Programmierer sogar das Pragma an sich. Auf Messen fragt z. B. so mancher „Kann der Compiler auch Pragmas?" und meint damit, ob man Betriebssystemfunktionen mit den üblichen Pragma-Zeilen direkt anspringen kann. Außerdem kollidieren Pragmas nicht mit

dem C-Standard, da hier jeder Compiler alles kommentarlos überlesen muß, was er nicht kennt.

Bei den Include-Files, die mit MaxonC++ geliefert werden, hat „#pragma amicall" übrigens das Rennen gemacht, während die „extern Amiga"-Deklarationen in der vorliegenden Version nicht mehr auftauchen.

5.6 Noch mehr Features

5.6.1 Verbinden von Zeilen

Bekanntlich ist es in C voll Banane, wo Zeilen anfangen oder aufhören. Dagegen ist dies beim Preprozessor durchaus relevant, denn hier muß jede Anweisung genau eine Zeile umfassen. Natürlich kann es dazu kommen, daß eine Zeile, z. B. durch eine umfangreiche Makrodefinition, arg lang und unhandlich wird. Für solche Gelegenheiten gibt es den Backslash („Gegenschrägstrich"). Wenn ein „\" am Zeilenende steht, wird diese Zeile mit der nachfolgenden verbunden, z. B. so:

```
#define Eumel \
26731
```

Dies ist eine gültige Makrodefinition, denn „26731" wird hier in die vorhergehende Zeile gezogen. Dieses Verfahren läßt sich fortsetzen, so daß man beliebig viele Zeilen logisch zu einer einzigen verbinden kann.

5.6.2 Verbinden konstanter Zeichenketten

Wenn im Quelltext mehrere Zeichenketten unmittelbar aufeinander folgen, macht der Preprozessor eine einzelne daraus:

```
char *s = "Dritte Tür links, " "jeder nur ein Kreuz!";
```

Man benutzt dieses Feature vor allem dann, wenn ein Teil einer konstanten Zeichenkette in Form eines Makros vorliegt, z. B.

```
#define DEVICE "dh1:"
#define SUBDIR "cpp"
#define FILE "M+Plus"
```

```
char Dateiname[] = DEVICE SUBDIR "/" FILE;
```

Dies ist absolut äquivalent zu und identisch mit

```
char Dateiname[ ] = "DH1:cpp/M+Plus";
```

Dieses Verbinden von Zeichenketten ist, wie Sie hier sehen, der so ziemlich letzte Arbeitsschritt des Preprozessors und erfolgt somit nach dem Ersetzen von Makros.

5.6.3 Trigraph-Sequenzen

Angeblich gibt es immer noch Rechner, die keinen vollständigen ASCII-Zeichensatz besitzen. Mitr persönlich fallen da zwar nur der C64 und diverse CP/M-Kisten ein, aber trotzdem hat die ANSI-Kommision auch an solche Phantomrechner gedacht und eine Möglichkeit eingeführt, auch auf solchen Systemen standardkonform C zu implementieren: die Drei-Zeichen-Folgen oder Trigraph-Sequenzen, wie der Lateiner sagt.

Der Preprozessor ersetzt einige Zeichenfolgen, die jeweils mit zwei Fragezeichen beginnen, durch ein anderes Zeichen, das in C bzw. C++ gebraucht wird. Im einzelnen werden folgende Ersetzungen vorgenommen:

```
??=    #
??(    [
??)    ]
??/    \
??<    {
??>    }
??'    ^
??!    |
??-    ~
```

Das folgende Programm illustriert dieses Feature:

```
??=include ??/    <stream.h>

void main()
  ??< char c??(??)="Test";
  cout « c;   ??>
```

Ausgesprochen hübsch, nicht wahr? Da diese Ersetzungen ganz zu Anfang der Vorverarbeitung erfolgen, kann man auch Überraschungen erleben:

```
#include <stream.h>

void main()
{ cout « "(Nanu???)";
}
```

Dieses Programm gibt

```
(Nanu?]
```

aus, denn die entsprechende Trigraph-Sequenz wird rücksichtslos durch die Klammer „]“ ersetzt. Sollte man wirklich einmal in eine solche Verlegenheit kommen, kann man sie aber umgehen, indem man das oben erwähnte Verbinden von Zeichenketten benutzt, denn das geschieht erst lange nach dem Ersetzen der Drei-Zeichen-Folgen:

```
#include <stream.h>

void main()
{ cout « "(Nanu??" "?)";
}
```

5.6.4 Hausgemachte Fehlermeldungen

Die meisten Programmierer ärgern sich, wenn der Compiler einen Fehler meldet. Es gibt aber auch solche, die das dringende Bedürfnis haben, zusätzliche Fehlermeldungen selbst zu erzeugen, und die können sich freuen, denn C bietet für jeden etwas.

Eine Zeile der Form

```
#error Irgendwas
```

veranlaßt den Preprozessor, eine entsprechende Fehlermeldung auszugeben. Typische Anwendungen sind Tests, ob Makros definiert sind, etwa so:

```
#ifndef SYMBOL
#error SYMBOL ist nicht definiert!
#define SYMBOL 42  // Defaultwert
#endif

char c[SYMBOL];
int i = SYMBOL-1;
```

5.6.5 Getürkte Dateinamen und Zeilennummern

Tatsächlich werden nicht alle Programme vom Menschen geschrieben. Gerade C-Quelltexte werden oft von anderen Programmen erzeugt: Da gibt es Konvertierungsprogramme (im weitesten Sinne Compiler), die Pascal oder FORTRAN nach C umwandeln, nicht zu vergessen diese seltsamen C++-Frontends. Andere Beispiele sind Parsergeneratoren (ich bin übrigens stolz darauf, daß MaxonC++ vollständig ohne solche entstand - darum ist es ja auch so schnell), Oberflächengeneratoren wie MAXON's hervorragendes MakeAPP und andere Arten von CASE-Tools. Nehmen wir einmal an, wir haben ein Programm, in das auf der einen Seite eine Textdatei (also ein Quelltext im weitesten Sinn) hereinkommt und auf der anderen Seite ein C-Quelltext als Ausgabe erzeugt. Nun kann es natürlich sein, daß dieses C-Programm Fehler enthält, die auf eine fehlerhafte Eingabe zurückzuführen sind und deshalb vom Benutzer in der Eingabedatei korrigiert werden müssen. Dann wäre es natürlich hilfreich, wenn der C-Compiler tatsächlich die Fehlerposition in der Quelldatei melden würde. Oder man möchte das resultierende Programm gern mit einem Source-Level-Debugger testen, und dann möchte man natürlich nicht den C-Quelltext sehen (den man ja nicht selbst geschrieben hat), sondern direkt in der ursprünglichen Quelldatei debuggen.

Deshalb gibt es die Preprozessor-Anweisung „line", und zwar in zwei verschiedenen Formen.

```
#line 42
```

sagt dem Compiler, daß die folgende Zeile bitteschön die Nummer 42 haben soll, und

```
#line 4711 "Suck"
```

gibt die Pseudo-Zeilennummer 4711 und den Pseudo-Dateinamen „Suck" vor.

In den nachfolgenden Zeilen wird die so vorgegebene Zeilennummer ganz gewöhnlich fortgezählt. Auch der Pseudo-Dateiname bleibt erhalten, bis ein neuer angegeben wird.

5.6.6 Nix dahinter

Eine leere Preprozessor-Anweisung der Form

```
#
```

ist auch erlaubt und hat keine Bedeutung.

5.7 Vordefinierte Symbole

Der Preprozessor besitzt einige nützliche vordefinierte Makros. Dabei handelt es sich zum Teil um ANSI C-genormte Symbole, andere sind Bestandteil von C++ oder spezielle Eigenheiten von MaxonC++. Es ist nicht möglich, diese Namen umzudefinieren.

```
__LINE__
```

Das Makro „__LINE__" liefert jeweils die Zeilennummer, in der es benutzt wird, als dezimale „int"-Konstante. Entsprechend enthält

```
__FILE__
```

den Namen der aktuellen Quelltextdatei in Form eines Strings, z. B. so:

```
#include <stream.h>

void main()
{ cout « "Dies steht in Zeile " « __LINE__
  « " in der Datei " __FILE__ ".\n";
}
```

Man beachte, daß der Wert des Makros „__FILE__" eine konstante Zeichenkette ist und als solche natürlich auch mit davor oder dahinter stehenden Zeichenketten verbunden wird, so daß oben keinesweges zwei „«" vergessen wurden.

Praktisch sind auch die beiden Makros

```
__DATE__
```

und

```
__TIME__
```

die das Datum bzw. die Uhrzeit der Übersetzung angeben, und zwar vor allem dann, wenn man ein Programm mit einer eindeutigen Versionsangabe versehen will:

```
#include <stream.h>

void main()
{ cout « "Version 08.15/4711 vom " __DATE__ ", " __TIME__ "
Uhr\n";
}
```

Das Datum wird dabei in der für uns Europäer etwas ungewohnten Form „Monat Tag Jahr", z. B. „Mar 08 1992", geliefert, während die Uhrzeit ganz normal in der Form „Stunden:Minuten:Sekunden" vorliegt.

Das Makro

```
__STDC__
```

liefert in allen zu ANSI C kompatiblen Implementierungen den numerischen Wert 1 und soll ermöglichen, festzustellen, ob ein Compiler diesem Standard genügt, denn andernfalls ist „__STDC__" nicht definiert. Eine Anwendung:

```
#ifndef __STDC__
#error Tut mir leid, dieses Programm braucht einen ANSI C
Compiler
#endif
```

MaxonC++ nimmt sich die Freiheit, „__STDC__" sowohl im C- als auch im C++-Modus zu definieren, denn im wesentlichen ist C++ zu ANSI C abwärtskompatibel. Wie Sie sich denken können, ist „__STDC__" wie auch die vorhergehenden Makros Bestandteil des ANSI C-Standards, im Gegensatz zum folgenden:

```
__cplusplus
```

ist als leeres Makro definiert, und zwar nur bei C++-Compilern. Folglich ist es in MaxonC++ im ANSI-Modus unsichtbar, so daß man mit

```
#ifdef __cplusplus
```

leicht den aktuellen Modus testen kann. Dazu dient auch das folgende Makro, das es aber nur in MaxonC++ gibt:

```
__COMPMODE__
```

ist in MaxonC++ immer definiert, und zwar im C-Modus mit der „int"-Konstanten 0 und unter C++ mit 1. Außerdem möchte man oft auch noch gerne testen, um welchen Compiler (und ggf. welche Version) es sich handelt. Deshalb definiert MaxonC++ zusätzlich ein Makro namens

```
__MAXON__
```

das immer das zehnfache der Versionsnummer (z. B. 10 für Version 1.0) als ganzzahlige Konstante liefert:

```
#if !defined __MAXON__ || __MAXON__ < 10
#error Dieser Quelltext wurde für MaxonC++ ab V 1.0 geschrieben.
#endif
```

Die folgende Tabelle faßt noch einmal die Namen aller vordefinierten Symbole zusammen, einschließlich ihrer Werte und des Standards, durch den sie definiert sind:

__LINE__	ANSI C	Zeilennummer (int)
__FILE__	ANSI C	Dateiname (konstanter String)
__DATE__	ANSI C	Datum „Mmm tt jjjj“ (konstanter String)
__TIME__	ANSI C	Uhrzeit „hh:mm:ss“ (konstanter String)
__STDC__	ANSI C	1 __cplusplus C++ 2.0 - (kein Wert)
__COMPMODE__	MaxonC++	0 im C-Modus / 1 im C++-Modus
__MAXON__	MaxonC++	Versionsnummer, z. B. 10

5.8 Arbeitsweise des Preprozessors

Der Preprozessor von MaxonC++ arbeitet in einem einzigen Schritt, ohne irgendwelche Zwischenergebnisse abzuspeichern, und gibt seine Ausgabe auch direkt an den Compiler weiter. Trotzdem kann man sich den Preprozessor logisch als eine Reihe unabhängiger Programme vorstellen, die nacheinander den Quelltext bearbeiten und an das jeweils nachfolgende Programm weitergeben. Diese logischen Phasen der Vor-Verarbeitung sind im einzelnen die folgenden:

1. Trigraph-Sequenzen (5.6.3) werden ersetzt.

2. Endet eine Zeile mit einem Backslash „\“, wird sie mit der jeweils nachfolgenden Zeile verbunden (5.6.1).

3. Der Programmtext wird in eine Folge von Token zerlegt, wobei Kommentare durch Leerzeichen ersetzt werden.

4 Preprozessor-Anweisungen aller Art werden ausgeführt.

5. Makros werden expandiert.

6. Aufeinanderfolgende konstante Zeichenketten werden verbunden (5.6.2).

Diese Reihenfolge logisch streng voneinaner getrennter Operationen hat einige Folgen, die die Leistungsfähigkeit des Preprozessors zwar teilweise mindern, aber andererseits

seine Arbeitsweise überschaubarer machen. Zum Beispiel ist folgendes nicht erlaubt, da die Ausführung von Preprozessor-Anweisungen logisch vor der Makro-Ersetzung erfolgt:

```
#define INC #include

INC <stdio.h>
```

Es ist möglich, Zeilen mit der Trigraph-Sequenz „??/“ statt „\“ zu verbinden, aber andererseits nicht möglich, „\“ als Bestandteil einer Makrodefinition zu verwenden, und die Liste solcher Möglichkeiten und Einschränkungen ließe sich noch beliebig fortsetzen. Ich hoffe aber, daß ungefähr klar geworden ist, was es mit dieser logischen Reihenfolge auf sich hat.

Damit wäre die C- und C++-Sprachbeschreibung auch schon komplett. Das folgende Kapitel gibt noch Informationen zu einigen Aspekten der Amiga-Programmierung unter MaxonC++.

6. Was Sie als Amiga-Programmierer wissen sollten

6.1 Die Schnittstelle zum Betriebssystem

MaxonC++ wird mit den jeweils aktuellsten von Commodore herausgegebenen Amiga-Includedateien geliefert. Daran mußten nur geringfügige Modifikationen vorgenommen werden, da in diesen Dateien an einigen wenigen Stellen z. B. die C++-Wortsymbole „class" oder „template" als Bezeichner benutzt werden. Ferner unterstützt MaxonC++ Aufrufe von Betriebssystemfunktionen ausschließlich über direkte Einsprünge, da es in den Bibliotheken keine „glue functions" gibt. Deshalb wurden in den Prototyp-Dateien („clib/xxx_protos.h") die „#include"'s für die jeweiligen „pragma"-Dateien („pragma/xxx_lib.h") gleich eingebaut. Außerdem wurde das längste Includefile, „intuition/intuition.h", leicht gestrippt, d. h. ein großer Teil der Kommentare wurde entfernt. Dadurch konnten gleichermaßen der Platzbedarf des Systems und die Übersetzungszeit gesenkt werden. Der Verlust einiger Kommentare scheint dagegen verschmerzbar zu sein, zumal jedes Amiga-Programm eine graphische Benutzeroberfläche haben sollte und „intuition/intuition.h" deshalb zu den meistbenutzten Includefiles zählt.

Im Prinzip ist das Amiga-Betriebssystem objektorientiert konzipiert, nur daß es eben in normalem C geschrieben wurde. Es würde sich in der Tat anbieten, für MaxonC++ eine wirklich objektorientierte Version der Includefiles zu erstellen, in denen dann z. B. „Library" nicht mehr „Node" als Member enthält, sondern davon abgeleitet wird. Viele Programmierer werden jetzt enttäuscht sein, daß wir dies nicht getan haben, aber wir hatten dafür gute Gründe:

Erstens sind die originalen, nur geringfügig an MaxonC++ angepaßten Includedateien naturgemäß die einzige kompatible Möglichkeit, den Amiga zu programmieren. Die zahlreichen Publikationen, die sich mit der Amiga-Systemprogrammierung befassen, beziehen sich ausnahmslos auf diese Standard-Definitionsdateien.

Zweitens veröffentlichte Commodore schon während der Entwicklung der 1.0-Version von MaxonC++ zweimal neue Includes, und es ist nicht anzunehmen (und auch nicht zu hoffen, denn das wäre das Ende des Amiga), daß die vorliegenden Includes endgültig bleiben werden. Eine Maxon-eigene, objektorientierte Schnittstelle zum System müßte immer wieder an neue Betriebssystemversionen angepaßt werden, was für Sie jedesmal eine empfindliche Verzögerung bedeuten würde.

Die „Intuition Classes" unterscheiden sich zudem in ihrem internen Layout so deutlich von C++-Klassen, daß es nicht möglich war, hier eine direkte Schnittstelle zu implementieren.

Für zukünftige Versionen ist aber geplant, Teilbereichen des Amiga-Betriebssystems ein zusätzliches objektorientiertes Interface aufzusetzen, das aber immer nur eine Ergänzung zur direkten Systemprogrammierung darstellen kann.

Als kleine Besonderheit ist es nicht nötig, die folgenden Libraries zu öffnen:

Name der Library	Basisvariable
exec.library	SysBase
dos.library	DosBase
diskfont.library	DiskFontBase
graphics.library	GfxBase
intuition.library	IntuitionBase

Die Exec-Library ist sowieso immer offen, DOS wird vom Startup-Code für die Standardein- und -ausgabe geöffnet, und wenn „GfxBase", „IntuitionBase" oder „DiskFontBase" im Programm referiert werden, bindet MaxonC++ automatisch Code ein, der diese Bibliotheken öffnet und am Programmende wieder schließt. Natürlich schadet es auch nicht, wenn ein Programm beides zusätzlich macht.

6.2 Start von der Workbench

Es gibt bekanntlich zwei verschiedene Arten, auf dem Amiga ein Programm zu starten, nämlich vom CLI (bzw. einer Shell) aus oder von der Workbench. Offensichtlich unterscheiden diese beiden Arten sich erheblich:

- Zu jeder Shell gehört ein Text-Window, in das ein Programm Ausgaben machen und aus dem es Eingaben lesen kann. Dieses Fenster entspricht in C++ der Standardein- und -ausgabe. Dagegen gibt es beim Start von der Workbench zunächst kein solches Window, so daß auch die Standard-I/O-Kanäle undefiniert sind.

- Beim Shell-Start werden dem Programm Argumente in der Kommandozeile übergeben und stehen über „argc" und „argv" direkt zur Verfügung, während ein von der Workbench gestartetes Programm Argumente in Form aktivierter Icons (und ihrer „Tool Types") besitzt.

Daraus folgt natürlich, daß beide Arten des Programmstarts ganz unterschiedlich gehandhabt werden müssen. Der Startup-Code vom MaxonC++ prüft zunächst, ob das Programm von der Workbench oder vom CLI aufgerufen wurde. Beim CLI-Start springt es wie gewohnt in die Funktion „main" ein, während im Falle eines Starts von der Workbench eine Funktion namens „wbmain" aufgerufen wird.

Moment mal, Sie haben doch bisher nie eine Funktion dieses Namens definiert? Das macht nichts, denn wenn der Linker kein „wbmain" findet, nimmt er eben das vordefinierte aus den Libraries, und diese Funktion macht überhaupt nichts. Also terminiert ein Programm ohne eigenes „wbmain" sofort, wenn es von der Workbench gestartet wird.

Um ein Programm also von der Workbench aufrufbar zu machen, muß man eine eigene „wbmain"-Funktion deklarieren, und zwar im C-Linkmodus, da der Linker ein Symbol namens „_wbmain" erwartet. Als Parameter kann man optional einen Zeiger auf die Startup-Message (deren Typ „struct WBStartup" in <workbench/startup.h> definiert ist) deklarieren. In C sieht der Prototyp also folgendermaßen aus:

```
void wbmain(struct WBStartup *ws)
```

...und in C++ so:

```
extern „C" void wbmain(WBStartup *ws)
```

Diese Startup-Message enthält einige interessante Informationen, insbesondere den Zähler „sm_NumArgs", der die Anzahl der aktivierten Icons angibt, und den Zeiger „sm_ArgList", der auf einen entsprechend großen Vektor des Typs „struct WBArg" verweist.

Jedes „WBArg"-Element enthält den Namen der Datei („wa_Name") sowie ein Lock auf das Verzeichnis („wa_Lock"), in dem die jeweilige Datei liegt. Um eine der Dateien zu öffnen, muß man also erst ihr Verzeichnis zum aktuellen Directory machen.

„sm_NumArgs" ist immer mindestens 1, und das erste (oder besser gesagt „nullte") Element des Vektors „sm_ArgList" repräsentiert Name und Verzeichnis des Programms selbst.

Das folgende Programm demonstriert einen typischen Programmstart auf dem Amiga einschließlich Auswertung der Argumente:

```
#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>
#include <workbench/startup.h>
#include <stdlib.h>

#define WW 640          // Fensterbreite
#define WH 200          // Fensterhöhe
```

```
NewWindow mynewwin =
{ 0, 0, WW,WH,
  2, 1, CLOSEWINDOW,
  ACTIVATE | WINDOWSIZING | WINDOWDRAG | WINDOWDEPTH |
  WINDOWCLOSE,
  NULL, NULL, "Startup-Demo",
  NULL, NULL,   100, 50, 640, 200,
  WBENCHSCREEN
};

struct Window* mywin = 0;

void FensterOeffnen() // Öffnet ein Fenster und bricht ab,
                      // falls das nicht klappt
{ mywin = OpenWindow(&mynewwin);
  if (!mywin)
    exit(1000);
}

void WartenUndSchliessen()
{ // Auf Benutzer-Aktion, insbesondere Close-Gadget, warten:
  WaitPort(mywin->UserPort);

  // Fenster schließen:
  CloseWindow(mywin);
}

void TextAusgabe(char *string, int x, int y)// String an
        // Pixel-Position x/y ausgeben
{ IntuiText it = { 1, 0, // FrontPen, BackPen
     JAM1,     // DrawMode
     0, 0,     // Offset x/y
     NULL,     // Font
     string,   // Text
     NULL };   // Next
  PrintIText(mywin->RPort, &it, x, y);
}
```

```
// ***** Shell-Start:

void main(int argc, char **argv)
{ int i;

  FensterOeffnen();

  TextAusgabe("Start vom CLI", 10, 20);
  for(i=0; i<=argc; i++)
    TextAusgabe( argv[i], 20, 30+9*i );

  WartenUndSchliessen();
}

// ***** Start von der Workbench:

#ifdef __cplusplus
  #define LINKAGE_C extern "C"
#else
  #define LINKAGE_C
#endif

LINKAGE_C void wbmain(struct WBStartup *ws)
{ int i;

  FensterOeffnen();

  TextAusgabe("Start von der Workbench", 10, 20);
  for(i=0; i < ws->sm_NumArgs; ++i)
    TextAusgabe( ws->sm_ArgList[i].wa_Name, 20, 30+9*i );

  WartenUndSchliessen();
}
```

Index

Symbole

! 24, 287
!= 89, 287
..... 319
..... 324
#define 320
#elif 328
#else 327
#endif 326
#error 335
#if 326
#ifdef 327
#ifndef 327
#pragma 329
#pragma + ... 329
#pragma - 329
#pragma amicall 332
#pragma break + ... 330
#pragma chip 129, 329
#pragma fast 129, 329
% 25, 287
%= 32, 287
& 29, 85, 127, 287
&& 29, 287
&= 32, 287
() 287
* 25, 83, 287
*** User break 331
*= 32, 287
+ 24, 26, 287
++ 22, 23, 287, 306
+= 32, 287
- 24, 26, 287
-- 22, 23, 287, 306
-= 32, 287
-> 287, 305
->* 287
/ 25, 287
/= 287
:: 190
< 89, 287
<= 89, 287
= 32, 287, 291
«= 32, 287
»= 32, 287

== 89, 287
> 89, 287
>= 89, 287
?: 30
[ ] 287
\ 333
^ 29, 287
^= 32, 287
__COMPMODE__ 338
__cplusplus 338
__DATE__ 337
__FILE__ 337
__LINE__ 337
__MAXON__ 338
__STDC__ 338
__TIME__ 337
| 29, 287
| ~ 287
|= 32, 287
|| 29, 287
~ 24, 29, 287

A

abgeleitete Klasse 204
abstrakte Klassen 220
abstrakten Datentypen 220
Abstraktion 57
Absturz 91
Addition 26
Adresse 83, 85
amicall 126
Amiga-Betriebssystem 125
ANSI C 74
ANSI C-Modus 329
Anweisung 43
argc 107
Argumente 60, 97, 107
Argumentliste 60
argv 107
Arrays 90
Asm 124
Aufzählungen 144
Ausdruck 17, 43
AUSSCHLIESSLICHES ODER 29
Auswertungsregeln 35
auto 116

B

Backslash 14, 333
Backspace 21

BASIC 310
Basisklasse 204, 245
Bedingte Übersetzung 326
bedingter Ausdruck 30
Betriebssystem 341
Bezeichner 13, 45
Bibliotheksfunktionen 77
binäres Suchen 143
Bit-Verknüpfungen 29
Bitfeld 163
bitweise Negation 24
Block 45
break 52, 330
Breakpoint 329
bsearch 144
Byte 83

C

calloc 170
case 55
Cast 39
char 19
Charkonstante 21
Chip-RAM 128
cin 16
class 195, 206
CLI 107
const 131, 191
continue 52
Copy-Konstruktor 250
cos 80
cout 12, 16, 308
ctype.h 79

D

Datenobjekt 15
default 55
Default-Argumente 71
Default-Konstruktor 248
Default-Parameter 71
Definition 66
Deklaration 66
Dekrement 22, 306
delete 166, 287, 294, 298
Destruktor 252
Division 25
do 49
double 33
Drei-Zeichen-Folgen 334

E

Elementlisten 108
Ellipse 73, 283
else 41
enum 145
Ergebnistyp 60
euklidischen Algorithmus 59
exit 80
exp 80
Exponent 34
extern 119

F

fabs 80
Fallunterscheidungen 40
Felder 90
Fließkommatypen 33
float 33
floor 80
for 48
Fragezeichen 21
free 171
friend 198
Funktion 57
Funktions-Spezifierung 137
Funktionsaufruf 301

G

Gegenschrägstrich 333
ggt 59
Gleitpunktzahlen 33
goto 51
Gültigkeitsbereich 51, 64, 156, 196
Gültigkieitsbereich 45

H

Helmut Kohl 155
Hexadezimal 20

I

Identifier 13
if 41
include 11, 325
Index 90
Indexoperator 303
Information Hiding 193
Initialisierung 39, 51, 100, 117, 234, 245
Inkrement 22, 306

Inline 136, 192
int 16
IntMult 32 26
isalpha 79
islower 79
isupper 79

K

Klasse 188
Klingelzeichen 21
Komma-Operator 33
Kommentar 10
konstante Zeichenkette 99
Konstruktor 241
Konvertierungsfuntionen 270
Kopier-Konstruktor 245, 250

L

L-Wert 22, 31
Label 51
Library 125, 342
line 336
Link-Spezifikationen 122
Linker 122
Literal 19
Literale 14
log 80
logische Negation 24
Logische Operatoren 29
long doubl 33
long int 18
long long int 18

M

main 12
Makrodefinition 320
malloc 170
Mantisse 34
math.h 80
Mehrfach-Vererbung 206, 246
Mehrfachvererbung 225
Member 150
Member-Funktion 189
Modul 118
modulo 25
Multiplikation 25
Multiplikative Operatoren 25

N

namenlose Unionen 161
new 165, 287, 294, 295
Nullbyte 99
Nullzeiger 89

O

ODER 29
Oktal 20
Operator 17, 270, 286
Operatorfunktionen 288
overload 70

P

Parameter 59, 60
Parameter in Registern 127
Parameterliste 58
Pascal 124
Pointer 83, 209
Postfix-Operatoren 22
pragma 75, 126
Preprozessor 11, 319
printf 78
private 193
protected 195, 212
Prototyp 67
Prozessorregister 126
public 193

Q

qsort 142
qualifiziert 197
qualifizierten Bezeichner 218
Quicksort 142

R

rand 81
Random 81
Referenzen 111, 133
Referenztyp 111
Register 126
Register-Variablen 127
Rekursion 65
Rest 25
return 57, 61

S

scanf 79

Schleifenstrukturen 47
Schlüsselwort 13
Scope 45, 51, 156, 196, 211
Scope-Operator 190
Seitenvorschub 21
Semikolon 47
set_new_handler 166
Shift-Operatoren 27
short int 18
sin 80
size_t 24, 295
sizeof 24
Speicherklasse 127, 130
Sprünge 51
Sprungmarke 51
sqrt 80
Stack 116
static 116, 121, 162, 201
Statische Member 200
stdio.h 78
stdlib.h 80
strcmp 103
strcpy 101
stream.h 11
String 14, 98, 324
Stringkonstanten 21
Stringvergleiche 103
strlen 106
strncpy 102
struct 206
Struktur 150
Subklasse 204
Subtraktion 26
Suffix 20, 34
Superklasse 204
switch 55

T

Tabulator 21
tan 80
temporäre Objekte 112, 273
Token 321, 324
tolower 80
toupper 80
translation unit 118
Trigraph-Sequenzen 334
Triviale Typumwandlungen 279
Typbeschreibung 95
Typdefinition 130
typedef 130

Typkonvertierung 209
Typlabel 147, 154
Typname 147
Typsicheres Linken 122
Typwandlungen 39

U

Überladen 68
Überladene Funktionen 124, 140, 215, 277
Übersetzungseinheit 118
Unäre Operatoren 23
UND 29
undef 321
union 159
Unionen 159
unsigned int 17
unsigned long int 18
unsigned long long int 18

V

Variable 15
variablen Parameterlisten 73
Vektor 90, 97, 303
Verbund-Datentyp 150
Vererbung 189, 203
Vergleiche 28
Vielfachverzweigungen 54
virtual 216, 226, 254
Virtuelle Basisklassen 225
virtuelle Memberfunkion 216
void 12, 77
volatile 131, 134
völlig virtuelle Funktionen 220
Vorzeichen 24, 34

W

Wagenrücklauf 21
wbmain 342
Wertzuweisung 95, 291
while 48, 50
Workbench 342
Wortsymbol 13

Z

Zählschleife 48
Zeichenkette 14, 98, 333
Zeiger 83, 132, 138
Zeiger auf Member 266
Zeigertypen 87

Zeilennummer 337
Zeilentrenner 21
Zufallszahl 81
Zugriffs-Deklarationen 213
Zugriffsmodus 211
Zugriffsrechte 193
Zuweisungsoperator 32

Maxon C++

Referenz-Handbuch

MAXON computer

Inhaltsverzeichnis

1. Die ANSI C Includedateien 5

1.1 Verschiedene nützliche Funktionen: <stdlib.h> 5
1.1.1 Umwandlung zwischen Zahlen und Strings 5
1.1.2 Mathematische Funktionen 7
1.1.3 Zufallszahlen 9
1.1.4 Speicherverwaltung, Typen und Zeiger 10
1.1.5 Programmende 12
1.1.6 Kommunikation mit dem Betriebssystem 13
1.1.7 Sortieren und Suchen 13

1.2 Ein- und Ausgaben: <stdio.h> 14
1.2.1 Typen, Objekte und Konstanten 14
1.2.2 Öffnen und Schließen von Dateien 16
1.2.3 Strings und Zeichenketten 18
1.2.4 Formatierte Ausgabe 20
1.2.5 Formatierte Eingabe 24
1.2.6 Dateioperationen 27
1.2.6.1 Dateiende und Fehlerbehandlung 27
1.2.6.2 Puffern 28
1.2.6.3 Externe Dateien 30
1.2.6.4 Binäre Daten 31
1.2.6.5 Positionieren in Dateien 32
1.2.7 Sonstige Funktionen 34

1.3 Fehlersuche: <assert.h> 35

1.4 Zeichenklassen: <ctype.h> 36

1.5 Fehlernummern: <errno.h> 39

1.6 Ganzzahlige Grenzwerte: <limits.h> 41

1.7 Mathematische Funktionen: <math.h> 43
1.7.1 Umwandlung von Zahlen in Zeichenketten 43
1.7.2 Fließkomma-Berechnungen 45

1.8 Haarsträubende Sprünge: <setjmp.h> 49

1.9 Signale und Ereignisse: <signal.h> 52

1.10 Variable Argumentlisten: <stdarg.h> 55

1.11 Implementationsabhängige Definitionen: <stddef.h> 58

1.12 Zeichenketten und Speicherverwaltung: <string.h> 59

1.13 Datum und Uhrzeit: <time.h> 66

2. Bibliotheken für C++ 71

2.1 Ein- und Ausgaben in C++: <stream.h> und <iostream.h> 71
2.1.1 Ausgabe 71
2.1.1.1 „ostream" und der Operator „«" 71
2.1.1.2 Basen und Manipulatoren 73
2.1.1.3 Formatierte Ausgabe 76
2.1.1.4 Ausgabe binärer Daten 77
2.1.2 Eingabe 78
2.1.2.1 Texteingaben mit „»" 78
2.1.2.2 Eingabefunktionen auf „istream" 79
2.1.3 Weitere Funktionen 81
2.1.3.1 Die Schnittstelle zum ANSI C-Dateisystem 81
2.1.3.2 Fehlererkennung 82

2.2 Ströme und Dateien: <fstream.h> 83
2.2.1 Klassen für Dateien 83
2.2.2 Dateien öffnen und schließen 83
2.2.3 Fehlererkennung 85
2.2.4 Konstruktoren und Destruktoren 86
2.2.5 Puffer 87
2.2.6 Positionieren in Strömen 87

2.3 Ein paar wichtige Definitionen in <new.h> 89

2.4 Eine String-Library: <tools/str.h> 91
2.4.1 Die Klasse „String" 91
2.4.2 Member-Funktionen 92
2.4.3 Der Operator „+" auf Strings 94
2.4.4 Vergleiche 95
2.4.5 Ein- und Ausgabe 96

2.5 Interna von Maxon C++: <streamdefs.h> 97

2.6 Funktionen für originelle Linker-Optionen: <linkerfunc.h> 100

Index 101

1. Die ANSI C Includedateien

In diesem Abschnitt werden die Includedateien besprochen, die Bestandteil des ANSI C Standards sind. MaxonC++ deklariert darin aber auch einige zusätzliche eigene Funktionen und Symbole. Diese sind jeweils mit einem „[M]" gekennzeichnet.

1.1 Verschiedene nützliche Funktionen: <stdlib.h>

In der Include-Datei <stdlib.h> werden einige nützliche Funktionen für die unterschiedlichsten Zwecke, z. B. zur Umwandlung von Zahlen und für die Speicherverwaltung, definiert.

1.1.1 Umwandlung zwischen Zahlen und Strings

strtol

```
long int strtol(const char *string, char **end, int base)
```

Die Funktion „strtol" wandelt eine Zeichenkette in eine ganze Zahl um. Am Anfang von „string" darf Leerraum stehen und hinter der Zahldarstellung dürfen noch andere Zeichen folgen. Wird für „end" ein Wert ungleich Null angegeben, so wird ein Zeiger auf den nicht umgewandelten Teil der Zeichenkette (z. B. auf die Zeichen, die der Zahldarstellung folgen) in der Variablen, auf die „end" verweist, abgelegt. Der Parameter „base" gibt die gewünschte Basis an: bei einem Wert zwischen 2 und 36 wird die Zeichenkette mit dieser Basis umgewandelt; ist der Wert 0, hängt die Basis wie in C üblich vom Präfix der Zeichenfolge ab (Hex bei „0x" oder „0X", oktal bei „0" und andernfalls dezimal).

Wenn das Ergebnis den Bereich von „long int" über- oder unterschreitet, wird „LONG_MAX" bzw. „LONG_MIN" zurückgegeben und die Variable „errno" erhält den Wert „ERANGE" (siehe <errno.h>).

strtoul

```
unsigned long strtoul(const char *string, char **end, int base)
```

Diese Funktion ist analog zu „strtol", aber der String wird in ein vorzeichenloses Langwort umgewandelt und im Falle eines Fehlers wird die Konstante „ULONG_MAX" zurückgegeben.

strtod

```
double strtod(const char *string, char **end)
```

Auch diese Funktion wandelt den Anfang einer Zeichenkette in eine Zahl um und zwar in einen „double"-Wert. Wie bei „strtol" wird ein Zeiger auf den nicht umgewandelten Teil der Zeichenkette in „*end" abgelegt, sofern „end" nicht Null ist. Bei einem Überlauf ist das Ergebnis „HUGE_VAL", bei einem Unterlauf wird „0" zurückgegeben.

strtovl [M]

```
long long strtovl(const char *string, char **end, int base)
```

Diese Funktion entspricht „strtol", wandelt aber in einen 64 Bit breiten „long long int"-Wert um, und im Falle eines Überlaufs ist das Ergebnis „LONGLONG_MAX" bzw. „LONGLONG_MIN".

strtouvl [M]

```
unsigned long long strtouvl(const char *s, char **end, int base)
```

Auch „strtouvl" führt in gewohnter Weise die Umwandlung eines Strings durch und zwar nach „unsigned long long int". Beim Überlauf wird „ULONGLONG_MAX" als Ergebnis geliefert, sonst ist alles wie bei „strtol".

atoi

```
int atoi(const char *string)
```

„atoi" wandelt eine dezimale Ziffernfolge in den entsprechenden numerischen Wert um und entspricht damit „(int) strtol(string, 0, 10)".

atol

```
long atol(const char *string)
```

Auch diese Funktion ist nur eine vereinfachte Form von „strtol" und entspricht „strtol(string, 0, 10)".

atof

```
double atof(const char *string)
```

Noch eine Kurzform, diesmal aber im Fileßkommabereich: Ein Aufruf von „atof“ ist äquivalent zu „strtod(string, 0)“.

ERANGE

```
#define ERANGE 1000
```

Diesen Wert erhält die Variable „errno“, wenn bei „strtol“ oder den verwandten Funktionen ein Überlauf auftritt.

HUGE_VAL

```
#define HUGE_VAL 1.797693134862316E+308
```

Rückgabewert der Funktion „strtod“ im Falle eines Überlaufs.

1.1.2 Mathematische Funktionen

abs

```
int abs(int i)
```

„abs“ liefert den absoluten Betrag seines Arguments, also so etwas wie „(i >= 0? +i : -i)“.

labs

```
long int labs(long int l)
```

Genau wie „abs“, nur eben für „long int“-Werte. Bekanntlich sind in Maxon C++ die internen Darstellungen von „int“ und „long“ identisch (jeweils 32 Bit), so daß auch diese beiden Funktionen sich praktisch nicht unterscheiden.

vlabs [M]

```
long long int vlabs(long long int vl)
```

Noch eine Betragsfunktion, diesmal für „long long int“ und damit natürlich eine Spezialität von Maxon C++. Ganz schön nervig, daß man in ANSI C nicht überladen kann...

div

```
div_t div(int nenner, int zaehler)
```

Da bei den üblichen Divisionsalgorithmen gleichzeitig Quotient und Rest ermittelt werden, ist es natürlich Zeitverschwendung, diese beiden Werte in zwei getrennten Rechenschritten zu berechnen. Der Datentyp „div_t“ ist eine Struktur mit den Einträgen „quot“ für den Quotienten und „rem“ für den Rest.

ldiv_t

```
ldiv_t ldiv(long nenner, long zaehler)
```

Das Ganze noch einmal für „long“-Zahlen.

div_t

```
typedef struct {
int quot;
int rem;
} div_t
```

In Strukturen dieses Typs werden die Ergebnisse der Funktion „div“ zurückgegeben.

ldiv_t

```
typedef struct {
long quot;
long rem;
} ldiv_t;
```

ist die Ergebnis-Struktur der Funktion „ldiv“.

1.1.3 Zufallszahlen

RAND_MAX

```
#define RAND_MAX 0x7fff
```

Diese Konstante stellt das größtmögliche Ergebnis von „rand" dar.

rand

```
int rand( void )
```

„rand" liefert eine Pseudo-Zufallszahl im Bereich von 0 bis RAND_MAX. Maxon C++ verwendet hier das additive Kongruenzverfahren, so daß ein Startwert definiert werden muß und zwar mit der folgenden Funktion:

srand

```
void srand(unsigned int seed)
```

Da die wenigsten Amigas einen Hardware-Zufallszahlengenerator eingebaut haben (am besten solche Dinger, die den Zerfall von Isotopen benutzen, denn solche Vorgänge sind wirklich zufällig), berechnet „rand" natürlich nur Pseudo-Zufallszahlen nach einem bestimmten Algorithmus (s. o.). Dieser Algorithmus benötigt einen Startwert, aus dem die erste Zufallszahl berechnet wird (und dann jede weitere Zahl aus der vorhergehenden). „srand" setzt diesen Startwert und sollte deshalb vor der ersten Benutzung von „rand" aufgerufen werden.

Wenn Sie dabei einen konstanten Wert verwenden, etwa „srand(4711)", erhalten Sie natürlich auch immer dieselbe Folge von „Zufallszahlen". Deshalb ist es üblich, hier einen Wert zu wählen, der von der Systemzeit abhängt, z. B. „srand(time(0))".

Random [M]

```
unsigned Random(unsigned u)
double Random()
```

Dieser überladene Funktionsname ist eine Spezialität von Maxon C++ und kann auch nur benutz werden, wenn <stdlib.h> im C++-Modus compiliert wird. Mit einen ganzzahligen

Argument „u“ liefert „Random“ eine Zufallszahl im Bereich von 0 bis u-1, ohne Argument eine Fließkommazahl zwischen 0 und 1 (wobei 0 möglicherweise vorkommen kann, 1 aber nicht). Diese Funktionen benutzen ein ähnliches Verfahren wie „rand“, beziehen aber die aktuelle Rasterzeile des Videoausgangs in die Berechnung mit ein. Dadurch muß die Zufallszahlenfolge nicht initialisiert werden und ist deshalb wesentlich „zufälliger“ als „rand“.

1.1.4 Speicherverwaltung, Typen und Zeiger

malloc

```
void malloc(size_t size)
```

„malloc“ reserviert „size“ Bytes Speicher und gibt einen Zeiger darauf zurück oder 0, wenn kein Speicher angefordert werden konnte. „size_t“ ist ein ganzzahliger Datentyp, der unter Maxon C++ als „unsigned“ definiert ist und „zufällig“ auch der Typ des Ergebnisses von „sizeof“ ist.

Mit der Funktion „set_new_handler“ (siehe <new.h>) kann eine Funktion spezifiziert werden, die so lange immer wieder aufgerufen wird, bis genügend Speicher frei ist - oder die Funktion mit „exit“ o. Ä. aussteigt.

calloc

```
void *calloc(size_t anzahl, size_t size)
```

Diese Funktion entspricht „malloc“, reserviert aber Speicher für einen „anzahl“ Elemente großen Vektor des Elementtyps „size“.

free

```
void free(void *ptr)
```

„free“ gibt einen mit „malloc“, „calloc“ oder „realloc“ reservierten Speicherbereich wieder frei.

realloc

```
void *realloc(void *ptr, size_t size)
```

Der Speicherplatzbedarf eines Objekts, auf das „ptr" zeigt, wird auf „size" geändert und ein Zeiger auf das (möglicherweise verschobene) Objekt zurückgegeben. Natürlich muß es sich um ein Objekt handeln, das mit „malloc", „calloc" oder einem anderen „realloc"-Aufruf erzeugt wurde.

Maxon C++ verfährt dabei wie folgt:

- Ist die neue Größe mit der alten identisch, muß nichts getan werden, und der Zeiger „ptr" wird unverändert zurückgegeben.
- Wenn das Objekt „verkleinert" wird, wird die entsprechende Anzahl Bytes am Ende des bisherigen Objekts freigegeben und „ptr" wieder unverändert zurückgegeben.
- Wenn das Objekt vergrößert werden soll, versucht „realloc" zuerst, den angrenzenden Speicherbereich zu reservieren. Sollte dies gelingen, wird ebenfalls „ptr" zurückgegeben.
- Andernfalls reserviert die Funktion einen neuen Speicherbereich der Größe „size" und kopiert das alte Objekt an den Anfang dieses Speicherblocks. Dann wird das alte Objekt gelöscht und ein Zeiger auf das neue zurückgegeben.
- Kann kein neues Objekt eingerichtet werden, ist das Ergebnis 0 und, das Objekt, auf das „ptr" zeigt, bleibt unverändert.

size_t

```
typedef unsigned size_t
```

„size_t" ist der Datentyp, der in C und C++ überall da verwendet wird, wo es um die Größe irgendwelcher Objekte geht.

NULL

```
#define NULL 0
```

Eine Konstante, die manche Leute gern für Zeiger des entsprechenden Werts verwenden. Eine Definition „((void*)0)" wie in vielen anderen C-Systemen ist hier übrigens nicht möglich, da C++ keine direkte Konvertierung von „void*" in andere Zeigertypen kennt.

wchar_t

```
typedef int wchar_t
```

Der Datentyp von „langen“ Zeichenkonstanten, z. B.

```
wchar_t C = L'x';
```

1.1.5 Programmende

exit

```
void exit(int res)
```

Diese Funktion beendet das Programm und entspricht einem „return res“ aus der Funktion „int main()“. Mit „atexit“ eingehängte Funktione werden ausgeführt, Destruktoren für globale Variablen aufgerufen und Resourcen (Dateien, Speicher usw.) freigegeben. Das Ergebnis Null zeigt normalerweise an, daß das Programm erfolgreich ausgeführt wurde, und jeder andere Wert wird als Fehlernummer interpretiert.

atexit

```
int atexit(void (*funktion)(void))
```

teilt der Laufzeitbibliothek mit, daß die angegebene parameterlose Funktion am Programmende ausgeführt werden soll.

abort

```
void abort(void)
```

Das Programm wird gnadenlos abgebrochen, ganz ohne Destruktoren, „atexit“-Funktionen und Resourcen-Freigabe.

1.1.6 Kommunikation mit dem Betriebssystem

getenv

```
char *getenv(const char *name)
```

Die Funktion liefert den Wert der Environment-Variablen „name" und liefert einen Zeiger darauf oder Null, wenn keine derartige Variable existiert.

Diese Funktion benötigt mindestens die Kickstart-Version 2.0 (37.175) und liefert bei früheren Versionen stets Null.

system

```
int system(const char *command)
```

Die Funktion führt den in „command" beschriebenen CLI-Befehl aus und liefert dessen Rückgabewert als Ergebnis. Auch diese Funktion benötigt mindestens Betriebssystemversion 2.0 und liefert andernfalls immer „-1". Die Ausgaben der Funktion erfolgen über die Standardausgabe des Prozesses - und in der Entwicklungsumgebung MCPP ist dies bei Betriebssystemversion 1.2 oder 1.3 leider die Umgebung, aus der Maxon C++ einmal gestartet wurde, nämlich das entsprechende CLI-Fenster oder „Nil:" beim Start von der Workbench.AbBetriebssystemversion 2.0 werden die Ausgaben korrekt umgelenkt.

1.1.7 Sortieren und Suchen

qsort

```
void qsort(void *vektor, size_t anzahl, size_t size,
int (*compare)(const void *, const void *))
```

Diese Funktion sortiert den Vektor „vektor[0] ... vektor[anzahl-1]", dessen Elemente jeweils die Größe „size" haben, mit der angegebenen Vergleichsfunktion. In Abschnitt 2.3.4 des Tutorials wird „qsort" ausführlich beschrieben.

bsearch

```
void *bsearch(const void *obj, const void *vektor,
              size_t anzahl, size_t size,
              int (*compare)(const void *, const void *))
```

sucht im Vektor „vektor[0] ... vektor[anzahl-1]“, dessen Elemente jeweils die Größe „size“ haben, anhand der angegebenen Vergleichsfunktion ein Element, das mit „*obj“ übereinstimmt. Da die Suche binär erfolgt, muß der Vektor dafür (z. B. mit „qsort“) sortiert sein. Auch diese Funktion wird in Abschnitt 2.3.4 des Tutorials näher erläutert.

1.2 Ein- und Ausgaben: <stdio.h>

1.2.1 Typen, Objekte und Konstanten

FILE

```
typedef struct stream FILE
```

Der Datenstrom ist das Hauptkonzept der Ein- und Ausgabe in ANSI C und entspricht einer Datei, die aber nicht unbedingt wie eine Datei aussehen (also auf einer Diskette oder Festplatte liegen) muß, sondern auch z. B. ein Terminal, ein Window oder ein Drucker sein kann. Der Datentyp, der solche Strom-Objekte beschreibt, heißt „struct stream“ oder „FILE“.

std__in, std__out, std__err [M]

```
extern FILE std__in, std__out, std__err
```

Dies sind die Namen der drei Standard-Dateien: „std__in“ für Eingaben des Programms, „std__out“ für Ausgaben und „std__err“ für Fehlermeldungen. Auf dem Amiga sind der Ausgabe- und der Fehlerkanal (leider) identisch. Beim Programmstart sind diese Dateien geöffnet und initialisiert (auch wenn sie beim Workbench-Start ins Nirvana zeigen) und müssen am Programmende auch nicht geschlossen werden.

Die drei Datenströme müssen keineswegs in jedem C-System diese Namen tragen, sondern können durchaus anders heißen. Deshalb sollte man sie nie unter diesen Bezeichnern ansprechen, sondern nur mit den drei folgenden Makros:

stdin, stdout, stderr

```
#define stdin (&std__in)
#define stdout (&std__out)
#define stderr (&std__err)
```

Die meisten Funktionen, die etwas mit Ein- und Ausgabe von Daten zu tun haben, erwarten als Argument keinen Datenstrom, sondern einen Zeiger auf einen solchen. Deshalb gibt es diese drei Makros, die jeweils einen Zeiger auf eine Standard-Datei liefern.

NULL

```
#define NULL 0
```

Die allseits beliebte Konstante (für Zeiger) wird auch in dieser Include-Datei definiert.

size_t

```
typedef unsigned size_t
```

„size_t" ist der Standardname für den Typ, den Ergebnisse von „sizeof" haben.

EOF

```
#define EOF (-1)
```

Die Konstante „EOF" wird an verschiedenen Stellen zum Anzeigen des Dateiendes bzw. eines Fehlers benutzt.

1.2.2 Öffnen und Schließen von Dateien

fopen

```
FILE *fopen(const char *name, const char *modus)
```

öffnet die Datei des angegebenen Namens und mit dem angegebenen Modus, erzeugt dazu eine FILE-Struktur und gibt einen Zeiger darauf zurück. Kann die Datei nicht geöffnet werden, liefert die Fuktion Null.

Es gibt die folgenden Modi:

"r" Vorhandene Datei zum Lesen öffnen.

"w" Datei zum Schreiben öffnen bzw. erzeugen und dabei evtl. vorhandene Datei überschreiben.

"a" Existierende Datei zum Anhängen ans Dateiende öffnen bzw. neue Datei erzeugen und zum Schreiben öffnen.

"r+" Vorhandene Datei zum Lesen und Schreiben öffnen oder ggf. eine neue Datei erzeugen.

"w+" Eine neue Datei erzeugen (ggf. existierende Datei löschen) und zum Lesen und Schreiben öffnen.

"a+" Vorhandene Datei fürs Lesen und Schreiben öffnen (ggf. neue Datei anlegen), wobei Schreib-/Leseposition auf Dateiende gesetzt wird.

ANSI C unterscheidet zwischen Text- und Binärdateien. Die oben aufgeführten Modi öffnen eine Datei als Textdatei. Will man eine Binärdatei öffnen, ist ein „b" an die zweite oder dritte Stelle des Modus-Strings zu setzen, z. B. „rb", „rb+" oder „r+b".

Auf dem Amiga (wie auch unter UNIX) sind die beiden Dateimodi identisch, und so ist es egal, ob Sie das zusätzliche „b" angeben oder nicht. MS-DOS Computerund die MessyDOS-Kopie Atari ST dagegen müssen tricksen und bei Textdateien dem Programm vorgaukeln, daß die Folge „Linefeed+Return" aus nur einem einzigen Zeichen besteht.

fclose

```
int fclose(FILE *f)
```

Eine geöffnete Datei muß auch wieder geschlossen werden. Die Funktion „fclose" schließt eine Datei, die mit „fopen" geöffnet wurde, und gibt die zugehörige „FILE"-Struktur frei. Vorher werden natürlich alle Puffer geleert (wie bei „fflush"). Das Ergebnis ist Null oder „EOF" bei einem Fehler, aber auf dem Amiga kann beim Schließen einer Datei nichts schiefgehen.

freopen

```
FILE *freopen(const char *name, const char *modus, FILE *file)
```

leitet einen bereits geöffneten Datenstrom in eine andere Datei um. Die physikalische Datei, mit der „file" bisher verbunden war, wird geschlossen, eine neue Datei mit dem angegebenen Namen und Modus geöffnet und ein Verweis darauf in die vorhandene FILE-Struktur gepackt.

Diese Funktion dient vor allem dazu, die Standard-Datenströme „stdin", „stdout" oder „stderr" in eine andere physikalische Datei umzuleiten.

FILENAME_MAX

```
#define FILENAME_MAX 200
```

Der Amiga kennt es bekanntlich so gut wie keine Beschränkung der Länge von Dateinamen, aber genau diese maximale Länge soll „FILENAME_MAX" eigentlich angeben. Die Konstante wurde mehr oder weniger willkürlich mit 200 definiert, denn länger wird ein Dateiname einschließlich Pfad wohl kaum werden.

FOPEN_MAX

```
#define FOPEN_MAX 99999
```

Dank dynamischer Datenorganisation ist die Anzahl der Dateien, die Sie mit Maxon C++ öffnen dürfen, nicht begrenzt und die Konstante „FOPEN_MAX" deshalb ziemlich Banane. Begrenzt ist diese Anzahl lediglich dadurch, daß irgendwann der Speicher voll ist oder das Betriebssystem sich sonstwie beschwert.

1.2.3 Strings und Zeichenketten

fgetc

```
int fgetc(FILE *f)
```

liest ein Zeichen aus der Datei „f" und wandelt es zuerst in „unsigned char" und dann in „int" um, denn bekanntlich gibt es in C keine klare Trennung zwischen Zahlen und Zeichen. Wenn das Dateiende überschritten wurde oder sonst ein Fehler auftrat, wird „EOF" geliefert.

getc

```
int getc(FILE *f)
```

Diese Funktion ist witzigerweise mit „fgetc" identisch, allerdings dürfen Implementationen sie aus nicht nachvollziehbaren Gründen in Form eines Makros definieren. Wie erst im Schwinden der Dominanz des Signifikats die semantische Struktur der Horizontflucht, des steten Aufschubs der Signifikation, entsteht, so wird andererseits mit der Substitution des Signifikats durch den Funktionszusammenhang der Signifikanten die Horizonterfahrung als Motorik des ständigen Verweisens ungültig.

getchar

```
int getchar (void)
```

ist äquivalent zu „getc(stdin)", liest also ein Zeichen aus der Standard-Eingabe.

ungetc

```
int ungetc(int c, FILE *f)
```

Manchmal merkt man erst, daß man genug gelesen hat, wenn man zuviel gelesen hat. Die Funktion „ungetc" stellt ein bereits gelesenes Zeichen in eine Eingabedatei zurück, so daß es beim nächsten Lesevorgang erneut gelesen wird. Bei einer „echten" Datei kann man sich „ungetc" wie das Zurücksetzten des Lesezeigers um ein Zeichen vorstellen, aber bei einer interaktiven Eingabe geht das so natürlich nicht. Deshalb hat jeder Datenstrom einen Puffer, in dem solche „zurückgestellten" Zeichen aufgenommen werden. Es paßt aber immer nur genau ein Zeichen in diesen Puffer.

fgets

```
char *fgets(char *s, int n, FILE *f)
```

„fgets" liest eine Zeile mit höchstens „n-1" Zeichen aus „f" und legt sie einschließlich eines abschließenden Null-Bytes im String „s" ab. Als Zeilenende gilt entweder ein Linefeed-Zeichen (‚\n') oder das Dateiende (EOF). In beiden Fällen wird das jeweilige Zeichen aber nicht in die Stringvariable aufgenommen. Wurde das Dateiende bereits überschritten oder trat ein Fehler auf, gibt „fgets" Null, andernfalls „s" zurück.

gets

```
char *gets(char *s)
```

entspricht „fgets(s, STREAM_MAXSTRING, stdin)", liest also eine Zeile von der Standardeingabe.

STREAM_MAXSTRING [M]

```
#define STREAM_MAXSTRING 80
```

Diese Konstante gibt an wieviele Zeichen maximal von „gets" gelesen werden, ist aber nicht Bestandteil des ANSI C-Standards.

fputc

```
int fputc(int c, FILE *f)
```

Natürlich kann man nicht nur Daten lesen, sondern sie auch schreiben. „fputc" ist die wohl rudimentärste Ausgabefunktion und schreibt ein Zeichen „c" in eine Datei „f". Das Ergebnis ist „EOF" bei Fehler oder andernfalls das ausgegebene Zeichen.

putc

```
int putc(int c, FILE *f)
```

Ähnlich wie „getc" und „fgetc", ist diese Funktion mit „fputc" identisch, darf aber als Makro definiert sein. Sie wissen ja, erst im Schwinden der Dominanz des Signifikats entsteht...

putchar

```
int putchar(int c)
```

schreibt das Zeichen in die Standard-Ausgabe und ist damit identisch mit „putc(c, stdout)".

fputs

```
int fputs(const char *s, FILE *f)
```

Diese Funktion schreibt eine Zeichenfolge, die wie allgemein üblich mit einem Nullbyte enden muß, in eine Datei. Bei einem Fehler liefert sie „EOF" und sonst einen Wert >= Null.

puts

```
int puts(const char *s)
```

dient ebenfalls zur Ausgabe von Zeichenketten und ist äquivalent zu „fputs(s, stdout)".

1.2.4 Formatierte Ausgabe

printf

```
int printf(const char *format, ...)
```

Diese ungemein nützliche Funktion ist die Standard-Ausgabefunktion in C, jedenfalls für Textdateien. Im Prinzip wird dabei die Zeichenkette „format" ausgegeben, nur an bestimmten Stellen werden andere Daten eingesetzt. Diese „bestimmten Stellen" sind Umwandlungsanweisungen, die stets mit einem „%" anfangen, mit einem Buchstaben enden und dazwischen noch zusätzliche Argumente haben dürfen. Ein Beispiel:

```
int a=17, b = 4;

printf("Die Summe von %d und %d ist %d.\n", a, b, a+b);
```

Die Zeichenfolge „%d" weist „printf" an, aus den sonstigen Argumenten das nächste zu nehmen, als „int" zu interpretieren und dezimal auszugeben. Also erzeugt der obige Funktionsaufruf folgende Ausgabe:

```
Die Summe von 17 und 4 ist 21.
```

Natürlich gibt es nicht nur „%d", sondern eine ganze Fülle von Format-Anweisungen, die folgendes in der genannten Reihenfolge enthalten können oder müssen:

- Das erste Zeichen ist, wie bereits erwähnt, stets „%".

- Eine optionale Folge von Flags in beliebiger Reihenfolge: Ein „-" bewirkt eine linksbündige Ausgabe der Daten (nur sinnvoll, wenn eine Feldbreite angegeben wird, siehe unten), „+" sorgt dafür, daß eine Zahl auf jeden Fall mit Vorzeichen (also ggf. „+") ausgegeben wird, wogegen ein Leerzeichen veranlaßt, daß bei vorzeichenlosen Zahlen ein Leerzeichen an den Anfang gesetzt wird. Das Flag „0" füllt bei Zahlen-Ausgaben das Ausgabefeld mit Nullen statt mit Leerzeichen auf (ebenfalls nur sinnvoll, wenn Feldbreite gesetzt), und „#" hat bei einigen Ausgabeformaten eine besondere Wirkung: Hexadezimalen Ausgaben wird ein „0x", oktalen eine „0" vorangestellt, Fließkommaausgaben enthalten auf jeden Fall einen Dezimalpunkt und bei den Anweisungen „g" und „G" werden Nullen am Ende nicht unterdrückt. Welch' ein Satz!

- Die Feldbreite und zwar als dezimale Ziffernfolge. Die Ausgabe des umgewandelten Arguments erfolgt (sofern nicht anders spezifiziert) linksbündig in einem Feld, das mindestens diese Breite hat, wobei bei Bedarf mit Leerzeichen (oder Nullen, siehe oben) aufgefüllt wird.

- Die Genauigkeit, die mit einem „." anfängt (damit „printf" sie von der Feldbreite unterscheiden kann), ebenfalls als dezimale Ziffernfolge. Bei Zeichenketten legt dies die maximale Anzahl von Zeichen fest, bei ganzzahligen Ausgaben die minimale Anzahl von Ziffern (nach links wird dann mit Nullen aufgefüllt) und bei Fließkommazahlen die Anzahl der Dezimalstellen.

- Ein Buchstabe als Längenangabe: Bei „h" ist das Argument als „short" oder „unsigned short" zu betrachten, „l" weist es als „long int" aus und bei „L" ist es entweder „long double" oder „long long" oder „unsigned long long".

- Das Umwandlungszeichen, z. B. „d" für ganzzahlige dezimale Ausgaben.

Außer dem „%" am Anfang und dem Umwandlungszeichen am Ende sind alle genannten Argumente optional. Als Genauigkeit oder Feldbreite kann anstelle einer Ziffernfolge „*" gesetzt werden. In diesem Fall steht der fehlende numerische Wert als nächstes „int"-Argument in der Argumentliste.

Es gibt folgende Umwandlungszeichen:

Zeichen	Argumenttyp	Art der Ausgabe
d	int	Dezimal
i	int	Wie „d“
o	int, unsigned	Vorzeichenlos oktal, ohne „0“ am Anfang
x	int, unsigned	Vorzeichenlos hexadezimal ohne führendes „0x“ und mit kleinen Buchstaben
X	int, unsigned	Wie „x“, aber mit Großbuchstaben
u	unsigned	Dezimal ohne Vorzeichen
c	int	Einzelnes Zeichen als „unsigned char“
s	char*	String, der mit Nullbyte endet
f	double	Fließkomma-Ausgabe in der Form „[-]xxx.yyy“. Die Genauigkeit legt die Anzahl der Nachkommastellen fest, Default ist 6.
e	double	Exponentielle Darstellung „x.yyyyyeõzzz“, wobei die Genauigkeit wieder die Zahl der Nachkommastellen angibt (Default 6).
E	double	wie „e“, aber mit großen „E“: „x.yyyyyEõzzz“
g	double	Bei Argumenten, deren Exponent zwischen der Genauigkeit (z. B. 6 als Default) und -4 liegt, erfolgt die Ausgabe wie bei „f“, andernfalls wie bei „e“.
G	double	Entspricht „g“, aber es wird zwischen „f“ und „E“ gewählt.
p	void*	Hexadezimale Speicheradresse
n	int*	Bewirkt keine Ausgabe, sondern legt die Anzahl der bisher ausgegebenen Zeichen in der Variablen, auf die das Argument zeigt, ab.
%	-	Gibt ein Prozentzeichen aus.

Das Ergebnis von „printf“ ist die Anzahl der ausgegebenen Zeichen oder negativ, wenn ein Fehler auftrat. Bitte denken Sie stets daran, daß „printf“ eine Funktion ohne Netz und doppelten Boden ist und fehlerhafte Argumente bestenfalls zu chaotischen Ausgaben und schlimmstenfalls zu einem Systemabsturz führen können.

fprintf

```
int fprintf(FILE *f, const char *format, ...)
```

Diese Funktion arbeitet eigentlich genau wie „printf", aber die Daten werden nicht über den Standard-Ausgabestrom, sondern in die Datei „f" ausgegeben. Andersrum formuliert, ist „printf(irgendwas)" eine Abkürzung für „fprintf(stdout, irgendwas)".

sprintf

```
int sprintf(char *s, const char *format, ...)
```

Auch diese Funktion entspricht „printf", aber die Ausgaben werden nirgendwo wirklich ausgegeben, sondern im String „s" abgelegt und mit einem Nullzeichen abgeschlossen. Sie als Programmierer sind selbst dafür verantwortlich, daß der Ziel-String groß genug ist, denn andernfalls wird (wie in C nicht anders zu erwarten) gnadenlos über den Vektor hinaus geschrieben, und der Guru gerät ins Meditieren.

vprintf, vfprintf, vsprintf

```
typedef unsigned va_list;

int vprintf(const char *format, va_list arg);

int vfprintf(FILE *f, const char *format, va_list arg);

int vsprintf(char *s, const char *format, va_list arg);
```

Diese Funktionen sind absolut äquivalent zu ihren Entsprechungen ohne das „v" am Anfang, aber die Argumente, die sonst dem Formatstring folgen, werden hier durch eine variable Argumentenliste („va_list") übergeben. Bitte lesen Sie dazu auch das Kapitel über <stdarg.h> oder fragen Sie Ihren Arzt oder Apotheker.

1.2.5 Formatierte Eingabe

scanf

```
int scanf(const char *format, ...)
```

Was „printf" für die Ausgabe, ist „scanf" für die Eingabe. Der Formatstring ist ähnlich aufgebaut wie bei „printf", hat aber eine andere Bedeutung:

Leerzeichen oder Tabulatoren in der Formatzeichenkette werden ignoriert. Bei allen anderen Zeichen außer „%" erwartet „scanf", daß dieses Zeichen als nächstes in der Eingabe folgt (eventuell mit vorangehenden Leerzeichen) und liest es bzw. bricht ab, wenn das Zeichen nicht folgt. Wieder leitet „%" Umwandlungsanweisungen ein.

Einem „%" kann optional ein „*" folgen (dann werden zwar Zeichen gelesen und umgewandelt, aber nirgendwo abgelegt), oder eine Zahl, die die maximale Feldbreite festlegt, also die maximale Anzahl von Zeichen, die bei dieser Umwandlung gelesen werden sollen. Die optionalen Zeichen „h", „l" und „L" legen wie bei „printf" die Datenbreite fest.

Zu jeder Umwandlungsanweisung gehört in der restlichen Argumentliste ein Zeiger auf eine Variable, es sei denn, hinter dem „%" folgt ein „*" (siehe oben). Bei einer Umwandlung überliest „scanf" zunächst alle Zwischenraumzeichen einschließlich Zeilentrennern, liest eine Zeichenfolge ein, die der Umwandlungsanweisung entspricht, z. B. eine Ziffernfolge bei dezimal-ganzzahliger Umwandlung, wandelt diese um und legt das Ergebnis an der beschriebenen Stelle ab. Wenn ein Fehler auftritt, bricht „scanf" sofort ab.

Bei den bereits mehrfach erwähnten Umwandlungszeichen handelt es sich um die Folgenden:

Zeichen	Argument	Erwartete Eingabedaten und Umwandlung
d	int*	Dezimal, ganzzahlig
i	int*	Ganzzahlig, wahlweise dezimal, oktal (mit „0" beginnend) oder hex (mit „0x" oder „0X" am Anfang)
o	int*	Oktal, ganzzahlig
x	int*	Hexadezimal mit oder ohne „0x"

c	char*	So viele Zeichen, wie die Feldbreite angibt (Default ist 1), wobei Leerzeichen, Zeilentrenner o. Ä. nicht überlesen werden. Will man ein Zeichen lesen, vorher aber Zwischenraumzeichen überlesen, ist die Umwandlung „%1s“ zu wählen. Im Gegensatz zu „%s“ wird bei „%c“ kein Nullbyte angehängt.
s	char*	Überliest zuerst Leerzeichen und Tabulatoren und liest dann eine Folge von Nicht-Zwischenraum-Zeichen. Ein Nullbyte wird an das Ergebnis angehängt.
e,f,g	float*	Fließkommazahl in nahezu beliebigem Format (mit oder ohne Voerzeichen, mit oder ohne Exponent, der ein großes „E“ oder ein kleines „e“ haben darf usw.)
p	int*	Hexadezimale Zahl
n	int*	Liest nichts, sondern legt im Argument die Anzahl der bisher gelesenen Zeichen ab.
[...]	char*	Längste Zeichenkette, die ausschließlich aus den Zeichen zwischen „[„ und „]“ besteht, z. B. Ziffernfolgen bei Umwandlungsoperator „%[0123456789]“. Auch hier kann die Länge durch Setzen einer Feldbreite beschränkt werden. Will man das Zeichen „]“ aufnehmen, ist es an den Anfang zu setzen (z. B. „%[ ]abcde]“). Die gelesenen Zeichen werden wie üblich mit einem ‚\0‘ abgeschlossen.
[^...]	char*	Liest die längste Zeichenkette, deren Zeichen sich nicht in der angegebenen Menge befinden.
%	-	Erwartet ein „%“, legt es aber nirgendwo ab.

Natürlich kann statt eines Zeigers auf „int“ stets ebenso gut ein Zeiger auf „unsigned int“ verwendet werden. Bei ganzzahligen Umwandlungsoperationen wird das zugehörige Zeigerargument durch die Längenangabe ‚h‘ als „short“ betrachtet, ‚l‘ macht daraus „long“ und ‚L‘ „long long“. Bei den Fließkommaoperationen ist ‚l‘ für „double“-Variablen und ‚L‘ für „long double“-Variablen zu setzen.

Beispiel:

```
int i,j;
short s;
char c[80];

scanf("Test %d %i, %hi %79s", &i, &j, &s, c)
```

akzeptiert z. B. die Eingabe

```
Test 42 0x686b, 99 Rest
```

oder auch

```
Test42 26731     ,99Rest
```

Das Ergebnis von „scanf" ist in beiden Fällen 4, und in die Variablen werden die Werte „42", „26731", „99" und „Rest" eingelesen.

Mehr noch als „printf" ist „scanf" eine höchst unsichere Funktion, da keine wirkliche Typprüfung der Argumente vorgenommen wird. Wenn man „nur" ein „&" vergißt (und damit den Wert einer Variablen statt ihrer Adresse als Argument übergibt), ist der Hangup oft nicht mehr zu vermeiden.

fscanf

```
int fscanf(FILE *f, const char *format, ...)
```

Dies ist das Äquivalent zu „scanf" für beliebige Dateien.

```
scanf(arglist)
```

ist gleichbedeutend mit

```
fscanf(stdin, arglist)
```

wobei „arglist" hier nichts mit Niedertracht und Heimtücke zu tun hat, sondern eine Argument-Liste darstellen soll.

sscanf

```
int sscanf(char *s, const char *format, ...);
```

ist ebenfalls eine andere Form von „scanf". Hier wird die Eingabe aus dem String „s" gelesen. Beispielsweise wandelt

```
sscanf(s, "%i", &n)
```

die Ziffernfolge „s" in eine Zahl um und leistet dabei ähnliches wie

```
n = atoi(s)
```

1.2.6 Dateioperationen

1.2.6.1 Dateiende und Fehlerbehandlung

feof

```
int feof(FILE *f)
```

liefert einen Wert ungleich Null, wenn in der genannten Datei das Dateiende überschritten wurde.

ferror

```
int ferror(FILE *f)
```

Diese Funktion schaut nach, ob bei der Datei „f" bisher ein Fehler aufgetreten ist und vermerkt wurde, und liefert die Fehlernummer oder andernfalls Null.

clearerr

```
void clearerr(FILE *f)
```

löscht die Dateiende- und Fehlervermerke einer Datei, so daß anschließend „feof" und „ferror" jeweils Null ergeben.

1.2.6.2 Puffern

setvbuf

```
int setvbuf(FILE *f, char *buf, int mode, unsigned size)
```

Normalerweise gehen Ausgabeoperationen erheblich schneller, wenn man die Daten nicht in kleinen Kleckermengen (z. B. Zeichenweise) schreibt, sondern sie erst einmal in einem Puffer sammelt und dann auf einen Schlag schreibt. Das gilt vor allem unter den alten Amiga-Systemversionen mit diesem unsäglich lahmen BCPL-DOS, aber auch unter 2.0 und höchstwahrscheinlich auch allen folgenden Versionen. Der Grund ist ganz einfach der, daß bei jeder Ausgabeoperation ein erheblicher Overhead nötig ist: Das Programm ruft eine DOS-Funktion auf, das DOS gibt an das File-System weiter, welches wiederum ein Device befragt, und erst das betätigt dann (z. B.) den Controller, der dann mit der Festplatte kommuniziert... Dieser Overhead ist für jede Datengröße praktisch identisch, so daß es natürlich Zeit spart, wenn man das ganze Prozedere nur einmal für einen großen Datenblock durchexerziert und nicht für jedes Byte einzeln. Analog gilt das natürlich auch für Eingaben aus einer Datei.

Die Funktion „setvbuf" versieht eine Datei mit einem solchen Puffer. Man kann sie für jede Datei nur einmal aufrufen, und zwar unmittelbar nach dem Öffnen, d. h. vor jeder Ein- und Ausgabe oder sonstigen Operation. „buf" zeigt dabei auf einen hinreichend großenSpeicherbereich, beim Wert Null legt „setvbuf" selbstständig einen solchen Puffer an und löscht ihn beim Schließen der Datei auch wieder. Der Parameter „size" gibt die gewünschte Puffergröße an. Erfahrungsgemäß sind hier einige hundert Bytes genug und eine weitere Steigerung bringt nicht mehr viel, und „mode" gibt einen der im folgenden beschriebenen Puffermodi an. Das Ergebnis von „setvbuf" ist Null, wenn alles klar ging.

IOFBF

```
#define _IOFBF 1
```

Beim Modus „_IOFBF" puffert „setvbuf" vollständig, d. h. der Puffer wird immer komplett gefüllt, bevor eine neue Operation erfolgt.

IOLBF

```
#define _IOLBF (-1)
```

Bei Ausgaben wird der Puffer im Modus „_IOLBF" höchstens bis zum ersten Zeilenende gefüllt. Dieser Modus ist z. B. auch bei Bildschirmausgaben brauchbar und bringen sogar eine gewisse Beschleunigung.

IONBF

```
#define _IONBF 0
```

Es wird nichts gepuffert. Einen Aufruf von „setvbuf" mit diesem Modus kann man sich also eigentlich sparen.

setbuf

```
void setbuf(FILE *f, char *buf)
```

Dies ist eine Spar-Version von „setvbuf", bei der „buf" auf einen Puffer der Größe „BUFSIZ" zeigen. „setbuf" legt nicht selbstständig einen Speicherbereich der geforderten Größe an.

```
char b[BUFSIZ];
FILE *f;

setbuf(f, b);
```

ist äquivalent zu

```
setvbuf(f, b, _IOFBF, BUFSIZ);
```

Fehler können bei „setbuf" nicht auftreten, und somit ist das Ergebnis „void".

BUFSIZ

```
#define BUFSIZ 200
```

ist die Puffergröße, die bei „setbuf" angenommen wird.

fflush

```
int fflush(FILE *f)
```

klingt zwar irgendwie gestottert, wirkt aber in Wirklichkeit einer stoßweisen Stotter-Ausgabe (will sagen: Schreiben mit Puffer) entgegen: Die Daten, die sich gerade im Puffer der Datei befinden, werden geschrieben, und der Puffer wird gelöscht. Bei Eingabe-Dateien hat „fflush" keine Wirkung, und wenn als Argument Null gewählt wird, werden alle gerade geöffneten Dateien geflusht.

1.2.6.3 Externe Dateien

remove

```
int remove(const char *name)
```

entfernt die Datei dieses Namens, entspricht also dem CLI-Befehl „delete". Hat das geklappt, ist das Resultet Null, andernfalls die Fehlernummer.

rename

```
int rename(const char *oldname, const char *newname)
```

benennt die Datei „oldname" in „newname" um, oder versucht es zumindest. Im Erfolgsfall wird Null als Ergebnis geliefert und andernfalls eine Fehlernummer.

tmpnam

```
char *tmpnam (char s[L_tmpnam])
```

Manchmal steht man vor der Situation, daß man einige Daten in temporären Dateien ablegen will. Die Funktion „tmpnam" liefert dafür Dateinamen, die garantiert mit keiner existierenden Datei überein stimmen: Sie beginnen mit „t:TMP_", und im Rest der Zeichenfolge sind die Adresse der Taskstruktur, die laufende Nummer des „tmpnam"-Aufrufs, das Datum und die Uhrzeit kodiert.

Der Parameter „s" muß auf eine Stringvariable der Länge (mindestens) „L_tmpnam" zeigen, in der das Ergebnis abgelegt wird, oder Null sein, dann wird der Dateiname in einem internen Buffer abgelegt. In beiden Fällen wird als Ergebnis ein Zeiger auf den String geliefert. „tmpnam" erzeugt keine Datei, sondern schlägt nur einen Namen vor. Außerdem stehen höchstens „TMP_MAX" verschiedene Namen zur Verfügung, aber bei Maxon C++ sind das mehr als genug.

L_tmpnam

```
#define L_tmpnam 40
```

Die Konstante gibt an, wie groß ein String wenigstens sein muß, um das Ergebnis von „tmpnam" aufzunehmen.

TMP_MAX

```
#define TMP_MAX 0x10000
```

„tmpnam" kann TMP_MAX mal aufgerufen werden, bevor sie sich wiederholt. In Maxon C++ sind das offensichtlich mehr als genug mögliche Dateinamen, aber darauf sollte man sich nicht verlassen - man denke hier nur an MesSy-DOS, wo Dateinamen nur 8+3 Zeichen lang sein dürfen. Übrigens muß man sich ganz schön beeilen, wenn man mit „tmpnam" wirklich zweimal denselben Namen erhalten will, denn Maxon C++ codiert ja auch die Uhrzeit in den Namen ein.

tmpfile

```
FILE *tmpfile (void)
```

ist so etwas wie die Kombination von „tmpnam" mit „fopen" und einer Löschautomatik: Eine temporäre Datei wird mit dem Modus „wb+" geöffnet, und beim Schließen dieser Datei oder am Programmende wird sie automatisch wieder gelöscht. Wenn das Ganze ein Schlag ins Wasser war, liefert „tmpfile" Null, andernfalls einen Zeiger auf die Datei.

1.2.6.4 Binäre Daten

fread

```
unsigned fread(void *ptr,
unsigned size, unsigned n, FILE *f)
```

„fread" liest aus der Datei „f" in den Vektor „ptr" maximal „n" Objekte der Größe „size". Bitte fragen Sie nicht, warum man hier und bei „fwrite" erst „size" und dann „n" angeben soll, während es bei „calloc", „qsort" und „bsearch" genau umgekehrt ist. „fread" liefert als Ergebnis die Anzahl der gelesenen Objekte (nicht Bytes!), die natürlich keineswegs größer, aber durchaus geringer als „n" sein kann (z. B. wenn das Dateiende erreicht wurde).

fwrite

```
unsigned fwrite(const void *ptr, unsigned size,
unsigned n, FILE *f)
```

ist das Gegenstück zu „fread“ und schreibt „n“ Elemente der Größe „size“ aus dem Vektor „ptr“ in die Datei „f“. Resultat ist die Anzahl der geschriebenen Objekte, die im Falle eines Fehlers kleiner als „n“ ist.

1.2.6.5 Positionieren in Dateien

fseek

```
int fseek(FILE *f, long offset, int modus)
```

setzt den Schreib- bzw. Lesezeiger in einer Datei an eine andere Position. Es gibt drei Modi: „SEEK_SET“ geht an die angegebene absolute Position (Dateianfang ist Null), „SEEK_CUR“ verschiebt den Zeiger um den Offset, und bei „SEEK_END“ bezieht sich der Offset auf das Dateiende, also z. B. „-1“ für das letzte Byte der Datei. Die Funktion liefert im Falle eines Fehlers einen von Null verschiedenen Wert.

SEEK_CUR, SEEK_SET, SEEK_END

```
#define SEEK_CUR 0
#define SEEK_END 1
#define SEEK_SET (-1)
```

Dies sind die drei Modi, die bei „fseek“ angegeben werden können.

ftell

```
long ftell(FILE *f)
```

liefert die aktuelle Dateiposition von Datei „f“.

rewind

```
void rewind(FILE *f)
```

setzt die Datei „f" an den Anfang zurück und löscht das Fehlerflag. Damit ist „rewind(f)" eine Abkürzung für „fseek(f, 0, SEEK_SET)" mit anschließendem „clearerr(f)".

fpos_t

```
typedef int fpos_t
```

Der Datentyp „fpos_t" wird gebraucht, weil einige Implementierungen zwischen Text- und Binärdateien unterscheiden und die Anwendung „fseek" dann bei Textdateien etwas problematisch ist. Deshalb gibt es speziell für Textdateien die beiden folgenden Funktionen:

fgetpos

```
int fgetpos(FILE *f, fpos_t *posn)
```

„fgetpos" speichert die aktuelle Position der Datei „f" in „posn", insbesondere zur späteren Verwendung mit „fsetpos". Wenn das System selbst nicht so genau weiß, was diese Posotion ist, liefert „fgetpos" einen Wert ungleich Null.

fsetpos

```
int fsetpos(FILE *f, const fpos_t *posn)
```

Das versprochene Gegenstück zu „fgetpos": Die Datei wird wieder auf die Position gesetzt, die in „posn" mittels „fgetpos" abgespeichert wurde.

1.2.7 Sonstige Funktionen

perror

```
void perror(const char *s)
```

gibt eine Fehlermeldung aus, die sich zum einen aus dem String „s" und zum anderen aus der Fehlermeldung, die zu der Nummer gehört, die gerade in „errno" steht. Hat „errno" z. B. den Wert 205, gibt

```
perror("Das war wohl nichts")
```

aus:

```
Das war wohl nichts: Object not found
```

„errno" wird eigentlich immer gesetzt, wenn irgend etwas schief geht, z. B. wenn eine Datei nicht geöffnet werden kann. Näheres steht in <errno.h>.

exit

```
void exit(int res)
```

lesen Sie hierzu bitte in Kapitel 1.1.5 (stdlib.h) nach.

1.3 Fehlersuche: <assert.h>

assert

```
#ifdef NDEBUG
#define assert(C)
#else
extern "Asm" void do__assert(char*, char*, unsigned int);
#define assert(C) { if(!(C)) do__assert(#C, __FILE__, __LINE__);
}
#endif
```

Die Include-Datei „assert.h" definiert lediglich ein einziges Makro nämlich „assert". Man kann damit die Sicherheit seines Programms erhöhen, indem man Plausibilitätstests einführt. Wenn z. B. eine Funktion einen Parameter zwischen 0 und 42 erwartet, setze man an ihren Anfang folgendes:

```
int f(int i)
{
assert( i>0 && i<42 )

i++;        // usw.
```

Gilt die bei „assert" angegebene Bedingung nicht, steigt das Programm bei einem ungültigen Parameter mit einer Meldung wie

```
Assertion failed: i>0 && i<42, file "murx.cpp", line 4711
```

aus - es sei denn, man definiert das Makro „NDEBUG", bevor man es einbindet. In diesem Fall werden die „assert"-Anweisungen ersatzlos gestrichen und kosten so weder Laufzeit noch Speicherplatz. Man sollte „NDEBUG" immer erst dann definieren, wenn man der Meinung ist, das Programm sei fehlerfrei. Auf jeden Fall aber vermeide man Bedingungen mit Seiteneffekt:

```
FILE *fp;
assert(fp = fopen("Caddy.config", "r"))      // NEIN!
```

Diese Anweisung gibt zwar eine mehr oder weniger aussagekräftige Fehlermeldung ab, wenn das Programm seine Konfigurationsdatei nicht finden kann, und steigt dann aus, was durchaus sinnvoll ist - aber irgendwann kommt man vielleicht doch noch auf die Idee, „NDEBUG" zu definieren, und dann wird man sich wundern.

1.4 Zeichenklassen: <ctype.h>

In der Definitionsdatei <ctype.h> findet der Programmierer einige kleine, aber recht nützliche Funktionen, durch die Zeichen weitgehend unabhängig vom verwendeten Rechner und Zeichencode behandelt werden können. Dies ist enorm wichtig für saubere (d. h. portable) Programmentwicklung.

isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit

```
int isalnum (int c)
int isalpha (int c)
int iscntrl (int c)
int isdigit (int c)
int isgraph (int c)
int islower (int c)
int isprint (int c)
int ispunct (int c)
int isspace (int c)
int isupper (int c)
int isxdigit(int c)
```

Diese Funktionen testen, ob das Zeichen „c" zu einer bestimmten Zeichgenklasse gehört, und liefern dann den entsprechenden logischen Wert (also „0" oder „1"). Es gibt prinzipiell folgende Klassen von Zeichen:

- Großbuchstaben von ‚A' bis ‚Z'
- Kleinbuchstaben von ‚a' bis ‚z'
- Allgemeine Buchstaben, d. h. Groß- oder Kleinbuchstaben. Umlaute werden dabei jeweils leider nicht berücksichtigt.
- Dezimale Ziffern von ‚0' bis ‚9'
- Hexadezimale Ziffern: ‚0' bis ‚9', ‚A' bis ‚F' sowie ‚a' bis ‚f'
- Leerraum, d. h. Leerzeichen, Zeilentrenner (Linefeed), Tabulatoren, Rücklauf (Return) und Seitenvorschub
- Sichtbare Zeichen einschließlich des Leerzeichens - das sind auf dem Amiga die Codes von 0x20 bis 0x7F sowie von 0xA0 bis 0xFF.
- Steuerzeichen, d. h. das genaue Gegenteil der sichtbaren Zeichen

Die folgende Liste gibt an, welche Funktion bei welcher Zeichenklasse „wahr" liefert:

isalnum	Buchstabe oder Ziffer
isalpha	Buchstabe
iscntrl	Steuerzeichen
isdigit	Dezimale Ziffer
isgraph	Sichtbares Zeichen außer Leerzeichen
islower	Kleinbuchstabe
isprint	Sichtbares Zeichen
ispunct	Sichtbares Zeichen außer Leerzeichen, Buchstabe oder Ziffer
isspace	Leerraum-Zeichen
isupper	Großbuchstabe
isxdigit	Hex-Ziffer

tolower

```
int tolower(int c)
```

Die Funktion wandelt das Zeichen „c", sofern es sich dabei um einen Großbuchstaben handelt, in den entsprechenden Kleinbuchstaben um. Andernfalls wird das Argument unverändert zurückgegeben. Die Funktion entspricht dabei in etwa der Fallunterscheidung

```
if (c >= 'A' && c <= 'Z')
  return c+('a'-'A');
else
  return c;
```

Auch hier werden Umlaute leider nicht berücksichtigt.

toupper

```
int toupper(int c)
```

Analog zu „tolower" werden hier Kleinbuchstaben in Großbuchstaben umgewandelt und andere Zeichen unverändert belassen.

which_xdigit [M]

```
int which_xdigit(char c)
```

Wenn der Parameter „c“ eine hexadezimale Ziffer ist, wird der zugehörige numerische Wert zurückgegeben, andernfalls „-1“. Beispiele:

```
which_xdigit('7')    =     7
which_xdigit('b')    =     11
which_xdigit('C')    =     12
which_xdigit('+')    =     -1
```

1.5 Fehlernummern: <errno.h>

Während der Laufzeit eines Programms kann so einiges schiefgehen - ein guter Programmierer geht immer davon aus, daß alles schief geht, was irgendwie schiefgehen kann. Die Analyse von Laufzeitfehlern wird mit den Definitionen der Datei <errno.h> ein wenig erleichtert.

errno

```
extern int errno
```

In dieser globalen Variablen legen viele Standard-Funktionen in Falle eines Falles eine Fehlernummer ab. Beispielsweise wird hier die entsprechende DOS-Fehlernummer zugewiesen, wenn bei „fopen" keine Datei geöffnet werden konnte. Vor solchen kritischen Operationen sollte man deshalb „errno" auf Null setzen.

ERANGE

```
#define ERANGE 1000
```

Diese Fehlernummer wird „errno" zugewiesen, wenn bei einer Funktion ein Wertebereich überschritten wurde.

EUSRBRK [M]

```
#define EUSRBRK 900
```

Diese Konstante gibt den Rückgabewert eines Programms an, wenn es vom Benutzer abgebrochen wurde. Obwohl die Konstante in <errno.h> definiert wird, wird dieser Wert dabei nicht wirklich an „errno" zugewiesen und kann folglich auch nicht abgefragt werden. Vielmehr wird diese Konstante definiert, damit man mit einer Anweisung wie „exit(EUSRBRK)" ein derartiges Programmende simulieren kann.

EASSERT [M]

```
#define EASSERT 990
```

Genau wie „EUSRBRK" ist dies kein Wert, den „errno" annimmt, sondern der Rückgabewert eines Programms, und zwar dann, wenn bei einem „assert"-Makro ein Fehler entdeckt wird.

EFREEMEM [M]

```
#define EFREEMEM 996
```

Auch dies ist ein Rückgabewert und keine wirkliche Fehlernummer. Mit diesem Wert bricht ein Programm ab, wenn bei „free“ bzw. „delete“ ein ungültiger Speicherbereich freigegeben werden soll.

ENONUM [M]

```
#define ENONUM 1001
```

Dieser Fehlercode wird von Funktionen wie „atoi“ oder „strtod“ gesetzt, wenn ein String keine gültige Zahldarstellung ist.

ENOMEM [M]

```
#define ENOMEM 1002
```

„new“ und die „malloc“-Funktion setzen diese Fehlernummer, wenn kein Speicher reserviert werden konnte.

1.6 Ganzzahlige Grenzwerte: <limits.h>

Jede Implementierung kann mehr oder weniger frei festlegen, welchen Umfang die einzelnen Datentypen haben. In der Datei <limits.h> werden Konstanten definiert, die die spezifischen Daten eines C-Systems angeben und so standard-gemäße Programmierung erleichtern.

CHAR_BIT

```
#define CHAR_BIT 8
```

gibt die Anzahl der Bits in einem „char" an.

CHAR_MAX, CHAR_MIN, SCHAR_MAX, SCHAR_MIN, SHRT_MAX, SHRT_MIN, INT_MAX, INT_MIN, LONG_MAX, LONG_MIN, LONGLONG_MAX, LONGLONG_MIN

```
#define CHAR_MAX      127
#define CHAR_MIN      (-128)
#define SCHAR_MAX     127
#define SCHAR_MIN     (-128)
#define SHRT_MAX      0x7fff
#define SHRT_MIN      (-0x8000)
#define INT_MAX       0x7fffffff
#define INT_MIN       (-0x80000000)
#define LONG_MAX      0x7fffffff
#define LONG_MIN      (-0x80000000)
#define LONGLONG_MAX  0x7fffffffffffffffLL
#define LONGLONG_MIN  (-0x8000000000000000LL)
```

Zu jedem vorzeichenbehafteten Datentyp existieren je zwei Konstanten, die den minimalen bzw. maximalen Wert dieses Typs angeben. Dementsprechend enden die Namen dieser Konstanten auf „_MIN" oder „_MAX" und beginnen je nach Typ mit „CHAR", „SCHAR", „SHRT", „INT", „LONG" oder „LONGLONG".

UCHAR_MAX, USHRT_MAX, UINT_MAX, ULONG_MAX, ULONGLONG_MAX

```
#define UCHAR_MAX       255
#define USHRT_MAX       65535
#define UINT_MAX        0xffffffffU
#define ULONG_MAX       0xffffffffU
#define ULONGLONG_MAX   0xffffffffffffffffULL
```

Bei den vorzeichenbehafteten ganzzahligen Datentypen ist naturgemäß stets Null der kleinste darstellbare Wert. Deshalb existiert für diese Typen lediglich jeweils eine Konstante, die den höchsten Wert des Datentyps darstellt.

1.7 Mathematische Funktionen: <math.h>

1.7.1 Umwandlung von Zahlen in Zeichenketten

PARAMETER_BASE, PARAMETER_DIGITS [M]

```
#ifdef __cplusplus
#define PARAMETER_BASE short=10
#define PARAMETER_DIGITS short=0
#else
#define PARAMETER_BASE short
#define PARAMETER_DIGITS short
#endif
```

Die nachfolgend beschriebenen Funktionen sind allesamt Spezialitätenvon Maxon C++ und wurden durch Default-Argumente etwas komfortabler gemacht. Da ANSI C aber keine Default-Argumente kennt, werden in den Parameterlisten diese beiden Makros benutzt. In C++ kann man also das jeweilige Argument weglassen, in C muß man es setzen. Übrigens werden die Makronamen am Ende von <math.h> mit „#undef" wieder gelöscht, so daß im weiteren keine Kollisionen auftreten.

inttostr, uinttostr, vlongtostr, uvlongtostr [M]

```
char *inttostr (int i, char s[ ], PARAMETER_BASE)
char *uinttostr (unsigned u, char s[ ], PARAMETER_BASE)
char *vlongtostr (long long ll, char s[ ], PARAMETER_BASE)
char *uvlongtostr (unsigned long long ull, char s[ ], PARAMETER_BASE)
```

Diese vier Funktionen wandeln ihr jeweils erstes Argument in eineZeichenkette um und legen das Ergebnis im String „s" ab. Der (in C++ optionale) dritte Parameter gibt die gewünschte Basis, eine Zahl von 2 bis 36, an, wobei 10 Default-Wert ist. Die umgewandelte Zeichenkette wird natürlich mit einem Nullzeichen abgeschlossen und ein Zeiger auf „s" zurückgegeben. Sie als Programmierer sind selbst dafür verantwortlich, daß der String „s" lang genug für das Ergebnis ist.

doubletostr, floattostr [M]

```
char *floattostr (float f, char s[ ], PARAMETER_DIGITS)
char *doubletostr (double d, char s[ ], PARAMETER_DIGITS)
```

Mit diesen beiden Funktionen können Sie bequem und ohne Benutzung von „sprintf" eine Fließkommazahl in eine Zeichenkette verwandeln. Das Ergebnis mit einem abschließenden Nullbyte wird dabei im String „s" abgelegt (der natürlich hinreichend lang sein sollte), und der letzte Parameter, für den im C++-Modus der Default-Wert Null definiert ist, gibt den Modus und die Ziffernanzahl an:

Bei einer positiven Anzahl von Stellen entspricht diese der Zahl der Nachkommastellen:

```
doubletostr(17.4, s, 5)
```

liefert

```
s = " 17.40000",
```

während eine negative Anzahl die Gesamtzahl der Dezimalstellen bei Exponentialdarstellung angibt:

```
doubletostr(17.4, s, -5)
```

führt zum Ergebnis

```
s = " 1.7400e+001".
```

und beim Argument „0" (bzw. dem Default-Argument) wird für Zahlen zwischen 10000 und 0.1 die Festpunktdarstellung mit der minimalen Anzahl von Nachkommastellen und sonst die Exponentialdarstellung gewählt.

Im Gegensatz zu den zuvor beschriebenen ganzzahligen Formatierungsfunktionen wird bei „doubletostr" und „floattostr" positiven Zahlen ein Leerzeichen vorangestellt.

1.7.2 Fließkomma-Berechnungen

sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, exp, log, log10, sqrt

```
double sin (double x)
double cos (double x)
double tan (double x)
double asin (double x)
double acos (double x)
double atan (double x)
double sinh (double x)
double cosh (double x)
double tanh (double x)
double exp (double x)
double log (double x)
double log10 (double x)
double sqrt (double x)
```

Diese Funktionen stellen die wichtigsten mathematischen Funktionen dar. Offensichtlich sind ihr Argument und Ergebnis jeweils „double"-Werte. Die Funktionen haben folgende (meist offensichtliche) Bedeutung:

sin	Sinus
cos	Cosinus
tan	Tangens
asin	sin^-1 für Argumente zwischen -1 und 1
acos	cos^-1 für Argumente zwischen -1 und 1
atan	tan^-1, Ergebnis zwischen -pi/2 und +pi/2
sinh	Sinus Hyperbolicus
cosh	Cosinus Hyperbolicus
tanh	Tangens Hyperbolicus
exp	Exponentialfunktion e^x
log	Natürlicher Logarithmus
log10	Logarithmus zur Basis 10
sqrt	Quadratwurzel

Bei den trigonometrichen Funktionen werden Winkel natürlich im Bogenmaß (Radian) angegeben. Ein Gradwinkel „w" läßt sich mit der Formel

```
x = 3.14159265358979/180.0 * w
```

in Bogenmaß umrechnen.

pow

```
double pow (double x, double y)
```

Diese Funktion berechnet die Potenz x^y, wobei aber „x" positiv sein muß.

atan2

```
atan2(double x, double y)
```

berechnet den Arcustangens von y/x, d. h. den Winkel, der zum Cosinus „x" und dem Sinus „y" gehört. Das Ergebnis ist aus [-pi, +pi], und die Funktion ist besonders dann praktisch, wenn man kartesische in Polarkoordinaten umrechnen will.

floor

```
floor(double x)
```

Die Funktion „floor" rundet auf die nächstkleinere ganze Zahl ab (genaugenommen die größte ganze Zahl kleiner oder gleich x). Das Ergebnis ist nichtsdestotrotz ein „double"- und kein „int"-Wert.

ceil

```
double ceil(double x)
```

Analog zu „floor" rundet „ceil" ihr Argument auf.

fabs

```
fabs(double x)
```

„fabs" liefert den Betrag ihres Arguments „x".

ldexp

```
ldexp(double x,int n)
```

Mit dieser Funktion erzeugt man aus einer Mantisse „x" und einem Binär-Exponenten „n" eine Fließkommazahl „x * 2^n". „ldexp" ist vor allem dann nützlich und auch ausgesprochen schnell, wenn man einen „double"-Wert mit einer Zweierpotenz multiplizieren will.

frexp

```
double frexp(double x ,int *expo)
```

„frexp" ist das Gegenstück zu „ldexp", denn hier wird das Argument in eine normalisierte Mantisse im Bereich von 0.5 bis 1 und einen Exponenten zur Basis 2 zerlegt. Die Mantisse wird als Funktionswert zurückgegeben, während der Exponent in „*expo" abgelegt wird.

modf

```
double modf(double x, double *dp)
```

Die Zahl „x" wird in ihre Vor- und Nachkommastellen aufgeteilt. Die Nachkommastellen, also eine Zahl zwischen -1 und +1, sind das Resultat der Funktion, während der ganzzahlige Teil nach „*dp" geschrieben wird. Beide Ergebnisse haben das gleiche Vorzeichen wie „x".

fmod

```
double fmod(double x, double y)
```

Diese Funktion sollte nicht mit „modf" verwechselt werden, sie berechnet den Rest, der bei einer Fließkommadivision „x/y" bei ganzzahligem Quotienten auftritt.

pwr10 [M]

```
double pwr10(int i)
```

Für ganzzahlige „i" im Bereich von -308 bis +308 liefert „prw10" die Potenz 10^i, und zwar ziemlich schnell, denn die Werte werden dabei teilweise einer Tabelle entnommen.

expo10 [M]

```
int expo10(double x)
```

Diese Funktion liefert die größte ganze Zahl „n“, für die „prw10(n)“ nicht größer als „x“ ist, also den Exponententeil von „x“. Nicht ganz zufällig sind „pwr10“ und „expo10“ Abfallprodukte der Stringkonvertierungsfunktionen von Fließkommazahlen.

fpwr10 [M]

```
float fpwr10(int i)
```

„fpwr10“ ist das „float“-Gegenstück zu „pwr10“ und berechnet ebenfalls die Potenz „10^i“, aber als „float“-Wert.

1.8 Haarsträubende Sprünge: <setjmp.h>

setjmp, longjmp

```
int setjmp(jmp_buf buf)
void longjmp(jmp_buf buf, int num)
```

Die normalen „goto"-Sprünge können nur innerhalb einer Funktion herumspringen, und das ist oft nicht genug. Manchmal muß man von einer Funktion in eine andere zurückspringen, z. B. wenn irgendwo in einer tief verschachtelten Reihe von Funktionsaufrufen ein Fehler auftritt und die Programmfunktion, aber nicht das ganze Programm abgebrochen werden soll. Es wäre dabei meist sehr umständlich, hinter jeden der zahllosen Funktionsaufrufe eine Abfrage der Art „Ist-ein-Fehler- aufgetreten-wenn-ja-dann-Ende" zu setzen. Deshalb gibt es die beiden Funktionen „setjmp" und „longjmp", mit denen man direkt aus einer Funktion in eine andere, höher gelegene zurückspringen kann.

Man kann dabei keineswegs wild in irgendeine Funktion hineinspringen, denn dann wäre der Absturz nahezu garantiert. Vielmehr speichert „setjmp" die wichtigsten Daten eines Zustands, nämlich den Stackpointer, den Programmzähler und die übrigen Prozessorregister, in einem Puffer des Typs „jmp_buf" ab, und „longjmp" stellt eben diesen Zustand wieder her, aber nur dann, wenn die Funktion, in der „setjmp" aufgerufen wurde, bisher nicht beendet wurde.

Der Ablauf ist also immer der folgende: In einer Funktion „f1" wird „setjmp" aufgerufen, und damit merkt sich das Programm haargenau die Position dieses „setjmp"-Aufrufs. Nun kann aus „f1" eine andere Funktion „f2" aufgerufen werden, welche möglicherweise wiederum irgendwann eine Funktion „f3" aufruft, in der ein Aufruf von „f4" stattfindet... Jedenfalls befindet das Programm sich irgendwann in einer Funktion „fx", und dann wird mit „longjmp" der abgespeicherte Zustand wiederhergestellt: Die Funktion „fx" wird beendet und alle ihre Daten sowie die der anderen Funktionen bis einschließlich „f2" werden vom Stack entfernt. Die Programmausführung wird in „f1" an genau der Stelle, an der das „setjmp" steht, fortgesetzt.

Nach jedem „setjmp" sollte das Programm dann feststellen, was passiert ist: Wurde soeben „setjmp" ausgeführt und der Programmzustand abgespeichert, oder ist ein „longjmp" erfolgt? Im letzteren Fall befindet sich das Programm schließlich scheinbar auch direkt hinter dem Aufruf von „setjmp". Deshalb hat „setjmp" einen Ergebniswert, der im Prinzip immer Null ist, und der Funktion „longjmp" kann ein numerisches Argument übergeben werden. Beim „longjmp"-Sprung wird dem Programm dann vorgegaukelt, „setjmp" habe dieses Argument als Ergebnis geliefert.

Am besten wird die Sache wohl an einem Beispiel deutlich. Eine typische Anwendung von „setjmp“ und „longjmp“ sieht so aus:

```
#include <setjmp.h>

jmp_buf Ausstieg;

void f2()
{
  printf("Funktion f2:\n");

  longjmp(Ausstieg,17);

  printf("PANIC\n");   // Diese Stelle wird nie erreicht.
}

void f1()
{
  if(setjmp(Ausstieg) == 0)
  // Ergebnis 0: Es wurde wirklich bloß der Zustand gespeichert
  {
    printf("Jump gesetzt.\n");
    f2();
    // Diese Stelle wird nur erreicht,
    // wenn kein "longjmp" gemacht wird:
    printf("DOUBLE PANIC\n");
  }
  else
    // Ergebnis 17: longjmp wurde ausgeführt
    printf("Jump ausgeführt.\n");
}
```

Die Funktion „f1“ macht dabei folgende Ausgabe:

```
Jump gesetzt.
Funktion f2:
Jump ausgeführt.
```

„jmp_buf“ ist stets ein Vektortyp, weshalb bei „setjmp“ kein Adressoperator „&“ vor das Argument gesetzt werden muß. Ihnen dürfte (oder sollte) klar sein, daß diese Funktion in der Regel ziemlich hanebüchen implementiert wird. Deshalb darf man „setjmp“ nur in ganz bestimmten Zusammenhängen benutzen:

- Man darf auf „setjmp“ unäre Operatoren, z. B. „-“ oder „!“, anwenden.

- Das Ergebnis kann mit den Operatoren „==“, „!=“, „<„, „>“, „<=“ oder „>=“ mit einem konstanten numerischen Ausdruck verglichen werden.

- Der Ausdruck kann „einfach so“ als Anweisung stehen oder bei „if“, „while“, „do“ oder „switch“ als Bedingung verwendet werden.

Ein Ausdruck wie

```
printf("Ergebnis: %d\n", setjmp(x))
```

würde also unter den meisten Implementierungen (einschließlich Maxon C++) beim anschließenden „longjmp“ zu einem gnadenlosen Absturz führen. Selbst ein scheinbar so harmloses Konstrukt wie

```
int i;
i = setjmp(x);
if (i == 0)
{ usw.
```

ist nicht zulässig (auch wenn es bei Maxon C++ zufällig klappt).

Es gibt noch einige andere Einschränkungen: Da „longjmp“ ein echtes Low-Level-Konstrukt ist, das den Compiler praktisch heimtückisch überlistet, werden zwar die automatischen Variablen der solchermaßen „abgewürgten“ Funktionen vom Stack entfernt, aber keine Destruktoren aufgerufen. Nach der Rückkehr durch „longjmp“ müssen Sie auch damit rechnen, daß die Optimierungen des Compilers Ärger machen und automatische Variablen jener Funktion, die nicht als „volatile“ deklariert sind, undefinierte Werte haben.

jmp_buf

```
typedef int jmp_buf[_JMP_BUF_SIZE]
```

Dieser Datentyp repräsentiert einen geeigneten Puffer, in dem „setjmp“ einen Programmzustand abspeichern kann.

JMP_BUF_SIZE [M]

```
#define _JMP_BUF_SIZE 16
```

Eine kleine, unbedeutende Hilfskonstante, die die Größe des Vektortyps „jmp_buf“ angibt.

1.9 Signale und Ereignisse: <signal.h>

signal

```
void (*signal(int sig, void(*f)(int)))(int)
```

oder besser:

```
typedef void (*P2F)(int)
P2F signal (int sig, P2F f)
```

Die Funktion „signal" setzt eine Funktion, die aufgerufen werden soll, wenn ein bestimmtes Signal „sig" eintrifft. In der vorliegenden Version von Maxon C++ ist die Funktion „signal" weitgehend Makulatur, denn bisher wird ein Signal ausschließlich vom Programm selbst durch die Funktion „raise" gesendet. Der Parameter „sig" gibt die jeweilige Signalnummer - in der Regel eines der im folgenden beschriebenen Makros - an, und „f" ist ein Zeiger auf eine Funktion, die ein „int" als Argument erhält und kein Ergebnis (also „void") liefert. Das Ergebnis von „signal" ist ein Zeiger auf die Funktion, die vorher als Signal-Handler gesetzt war.

Anschließend wird die Funktion „f" stets beim Eintreffen eines Signals „sig" aufgerufen und erhält dabei als Argument die Signalnummer, so daß eine Funktion durchaus als Handler für mehrere Signale verwendet werden kann.

Anstelle eines Zeigers auf eine Funktion kann auch eins der unten beschriebenen Makros „SIG_IGN", „SIG_DFL" oder „SIG_ERR" verwendet werden.

raise

```
int raise(int sig)
```

Die Funktion „raise" sendet das Signal „sig" und löst so den Aufruf des entsprechenden Handlers aus.

SIGTERM, SIGABRT, SIGPFE, SIGILL, SIGINT, SIGSEGV

```
#define  SIGTERM  0 // Normales Programmende
#define  SIGABRT  1 // Anormales Programmende wie bei "abort"
#define  SIGFPE   2 // Arithmetikfehler
#define  SIGILL   3 // Illegale Prozessor-Anweisung
#define  SIGINT   4 // Software-Interrupt
#define  SIGSEGV  5 // Segmentverletzung bei Speicherzugriff
```

Dies sind die sechs Standard-Signalnummern und damit die einzigen wirklich legalen Argumente für „signal“ und „raise“. Eine Implementierung darf aber auch noch zusätzliche eigene Signale haben.

SIG_IGN

```
#define SIG_IGN ((void(*)(int))0)
```

Wird bei „signal“ als Handlerfunktion „SIG_IGN“ angegeben, so wird das entsprechede Signal in Zukunft ignoriert.

SIG_DFL

```
#define SIG_DFL ((void(*)(int))1)
```

Diese Pseudo-Handlerfunktion stellt als Argument bei „signal“ den jeweiligen Default-Handler dar.

SIG_ERR

```
#define SIG_ERR ((void(*)(int))-1)
```

Die Funktion „signal“ liefert das Ergebnis „SIG_ERR“, wenn irgendein Fehler aufgetreten ist, z. B. die Argumente illegal waren.

sig_atomic_t

```
typedef int sig_atomic_t
```

Für echte Signal-Freaks gibt es den ganzzahligen Datentyp „sig_atomic_t“, der zur Kommunikation zwischen Programm und Handler benutzt werden kann. Die „besonde-

re“ Eigenschaft dieses Datentyps ist, daß ein Zugriff auf eine solche Variable eine unteilbare Operation ist, d. h. nicht unterbrochen werden kann. Wenn man einer Variablen dieses Typs einen Wert zuweist, kann man darauf vertrauen, daß ein währenddessen eintreffendes Signal entweder vor oder nach dieser Operation ausgeführt wird und nicht etwa mittendrin, wenn die Variable so etwas wie einen Zwischenwert hat.

Oder drücken wir es einmal so aus: Eine „long long“-Variable besteht unter Maxon C++ aus zwei Langworten, die bei einer Wertzuweisung nacheinander geschrieben werden. Hier könnte es durchaus vorkommen, daß ein Signal-Handler genau dann aufgerufen wird, wenn erst eines der beiden Langworte geschrieben ist. Der Wert der Variablen wäre dann während der Ausführung des Handlers irgendwie eine undefinierte Mischung aus dem alten und dem neuen Wert. Wenn man also in einer globalen Variablen Parameter an einen Handler übergeben möchte, sollte man dafür ausschließlich den Datentyp „sig_atomic_t“ benutzen.

1.10 Variable Argumentlisten: <stdarg.h>

va_list

```
typedef unsigned int va_list
```

Funktionen wie „printf" und „scanf" haben eine Parameterliste, in der jeweils nur die ersten Parameter einen Namen und Typ haben und anschließend beliebig viele weitere Argumente erlaubt sind. Syntaktisch wird dies in C++ durch die Ellipse „..." am Ende der Parameterliste deklariert, und man kann dieses Feature auch in selbstgeschriebenen Funktionen verwenden. Nun wird sich der geneigte Leser vielleicht fragen, wie eine Funktion denn solche namenlose Parameter mit nahezu beliebiger Anzahl und unterschiedlichsten Typen auswerten kann. Genau dabei hilft die Includedatei „<stdarg.h>", die einige Makros deklariert, mit denen man Stack-Argumente gleichermaßen komfortabel und portabel auswerten kann.

Das Prinzip beruht darauf, daß ein Zeiger hinter den letzten normal deklarierten Parameter gesetzt wird und dann der Reihe nach die restliche Argumentliste durchläuft. Dieser „Zeiger" wird repräsentiert durch eine Variable des Typs „va_list".

va_start

```
#define va_start(AP,LASTARG) (AP)=(unsigned int)(&LASTARG) \
+sizeof(LASTARG)
```

Ein Argumentlisten-Zeiger des Typs „va_list" wird normalerweise initialisiert, indem er auf die erste Adresse hinter dem letzten Parameter gesetzt wird, also auf genau die Speicheradresse, an der die formal unbekannten, restlichen Argumente anfangen. Dazu dient das Makro „va_start", z. B. so:

```
void fun(const char *format...)
{
  va_list v;
  va_start(v, format);

  // usw.
}
```

va_arg

```
#define  va_arg(AP,TYPE)  ((AP)+=sizeof(TYPE),  \
sizeof(TYPE)>1  ?  ((AP)=(AP)+1&(-2)):(AP),  \
((TYPE*)AP)[-1])
```

Das Makro „va_arg" mit seiner ziemlich wüsten Definition ist in seiner Anwendung um so einfacher: Ein Aufruf der Form

```
x = va_arg(vp, TYPE);
```

liest das nächste Argument des Typs „TYPE" aus der variablen Argumentliste, liefert es als Ergebnis und setzt nebenbei den Argumentzeiger „vp" entsprechend weiter. Wenn man also die auftretenden Datentypen kennt (bei „printf" gibt man diese ja im Formatstring an), kann man mit einer Folge von „va_arg"-Aufrufen nacheinander die Argumente lesen.

Beachten Sie aber, daß „kurze" ganzzahlige Typen (short, char) bei dieser Art der Wertübergabe automatisch nach „int" sowie „float" nach „double" umgewandelt werden. Deshalb ist z. B. „va_arg(x, unsigned short)" generell unsinnig und ist durch „va_arg(x, unsigned int)" zu ersetzen.

va_end

```
#define va_end(AP)
```

Bei einigen Implementierungen kann die Auswertung variabler Argumentlisten ein ziemicher Akt sein, und deshalb gibt es das Makro „va_end", mit dem man immer eine Argumentauswertung beenden sollte und das bei Bedarf irgendwelche Aufräumarbeiten durchführt. Bei Maxon C++ ist „va_end" ein leeres Makro, aber trotzdem sollte man es nie vergessen.

Zum Abschluß ein zusammenhängendes Beispiel für die Auswertung variabler Argumentlisten. Der Einfachheit halber ist die Argumentliste hier überhaupt nicht variabel, sondern es wird stets erwartet, daß ein „int"-, ein „double"- und ein „char"-Argument folgen. Trotzdem sollte das Prinzip klar werden:

```
#include <stdarg.h>
#include <stdio.h>

void fun(const char *format...)
{
  va_list v;
  int i;
  double d;
  char c;

  va_start(v, format); // initialisieren mit letztem bekannten
  // Parameter

  // die drei Argumente werden der Reihe nach gelesen:
  i = va_arg(v, int);
  d = va_arg(v, double);
  c = (char) va_arg(v, int); // Auch chars werden als „int"
                // Übergeben

  // der Ordnung halber aufräumen:
  va_end(v);

  printf("%s: %i , %g , %c\n", format, i, d, c);
}

void main()
{
  fun("Test", 42, 47.11, 'x');
}
```

1.11 Implementationsabhängige Definitionen: <stddef.h>

NULL

```
#define NULL 0
```

Die allgemein beliebte und geschätzte Konstante „NULL" wird auch in „<stddef.h>" deklariert.

offsetof

```
#define offsetof(s,m) ((unsigned)&((s*)NULL)->m)
```

In ANSI C gibt es keine Pointer auf Member. Als die C-Programmierer daraufhin neidisch nach C++ schielten, kamen sie offensichtlich auf die Idee, dieses Feature in C zu simulieren, indem sie den Offset eines Members innerhalb einer Struktur berechnen. Mit dem Makro „offsetof" wird dies erheblich erleichtert: „offsetof(s,m)" liefert den Offset (in Bytes) des Members „m" in der Klasse oder Struktur „s".

size_t

```
typedef unsigned size_t
```

„size_t" wird nicht nur in „<stdlib.h>" und „<stdio.h>", sondern auch in „stddef.h" definiert und stellt den Typen dar, den das Ergebnis von „sizeof" in einer bestimmten Implementation hat.

ptrdiff_t

```
typedef int ptrdiff_t
```

Subtrahiert man zwei Zeiger gleichen Typs voneinander, so ist das Ergebnis bekanntlich ein ganzzahliger Wert, der die Anzahl der Vektorelemente zwischen diesen Zeigern angibt. Nun kann der Typ dieser Differenz durchaus implementationsabhängig sein: Bei einigen Compilern muß eine Adressdistanz als „long int" ausgedrückt werden, bei anderen (wie Maxon C++) reicht auch ein „int". Deshalb gibt es für diesen Typ den standardisierten Namen „ptrdiff_t".

wchar_t

```
typedef int wchar_t
```

Noch eine Wiederholung: Der Name „wchar_t“ bezeichnet den Datentyp von langen Zeichenkonstanten.

1.12 Zeichenketten und Speicherverwaltung: <string.h>

strcpy

```
char *strcpy (char *s1, const char *s2)
```

Das Vektorkonzept von ANSI C ist bekanntlich „far beyond repair“ und hat unter anderem zur Folge, daß man Strings nicht mit einem einfachen „=“ zuweisen kann. Deshalb gibt es die überaus nützliche Funktion „strcpy“, die einen String „s2“, der mit einem Nullbyte enden sollte, in eine Stringvariable „s1“ kopiert. Das Ergebnis ist undefiniert, wenn die beiden Strings sich überlappen, und wie eigentlich immer haben Sie höchstpersönlich darüber zu wachen, daß die Zielvariable für den String „s2“ lang genug ist. Das Ergebnis von „strcpy“ ist der Zeiger „s1“.

strncpy

```
char *strncpy(char *s1, const char *s1, size_t n)
```

Die sicherheitsbetonte Variante von „strcpy“ kopiert maximal „n“ Zeichen aus „s2“ nach „s1“ und füllt den Rest mit Nullzeichen auf.

strcat

```
char *strcat (char *s1, const char *s2)
```

In diesem Fall heißt „cat“ nicht „Katze“, sondern ist eine Verbalhornung von „concat“ oder auf Deutsch „konkatinieren“ - man kann auch „aneinanderhängen“ dazu sagen. Die Funktion sucht sich das Ende des Zeichenvektors „s1“ (am obligatorischen Nullbyte unschwer zu erkennen) und kopiert den String „s2“ genau dorthin, wodurch die beiden Strings zusammengehängt werden. Auch hier ist es der Sorgfalt und Umsicht des Programmierers anheim gestellt, dafür zu sorgen, daß im String „s1“ auch noch genug Platz ist. Der Rückgabewert ist wieder einmal ein Zeiger auf das Ergebnis, also „s1“.

strncat

```
char *strncat(char *s1, const char *s2, size_t n)
```

Genau wie bei „strcat“ wird auch hier der String „s2“ an den Inhalt des Zeichenvektors „s1“ angehänht, aber hier werden maximal „n“ Zeichen kopiert.

strcmp

```
int strcmp (const char *s1, const char *s2)
```

Diese Funktion vergleicht die beiden Zeichenketten „s1“ und „s2“ und liefert:

- einen negativen Wert, wenn s1 < s2,
- Null, falls s1 == s2,
- eine positives Ergebnis, sofern s1 > s2.

Natürlich werden hier wirklich die Zeichen des Strings verglichen und nicht etwa die Adressen, wie es ja bei gewöhnlichen Stringvergleichen der Fall ist.

strncmp

```
int strncmp(const char *s1, const char *s2, size_t n)
```

„strncmp“ entspricht „strcmp“, vergleicht aber maximal die ersten „n“ Zeichen, oder auch weniger, wenn eher auf ein Nullzeichen und damit das Stringende gestoßen wird.

stricmp [M]

```
int stricmp(const char *s1, const char *s2)
```

Diese Funktion vergleicht die Zeichenketten „s1“ und „s2“ genau wie „strcmp“, wandelt aber zuerst alle Klein- in Großbuchstaben um, bevor zwei Zeichen verglichen werden. Diese Umwandlung geschieht natürlich nur intern, so daß es keinen Seiteneffekt auf die Strings „s1“ und „s2“ gibt. Beispielsweise liefert „stricmp(„Test“, „TEST“)“ Null.

strlwr [M]

```
char *strlwr(char *s)
```

In der durch ein Nullzeichen abgeschlossenen Zeichenkette „s" werden alle Großbuchstaben durch Kleinbuchstaben ersetzt. Ein Zeiger auf den solchermaßen veränderten String wird zurückgegeben.

strupr [M]

```
char *strupr(char *s)
```

funktioniert wie „strlwr", wandelt aber Klein- in Großbuschtaben um.

strchr

```
char *strchr (const char *s, int c)
```

Die Funktion sucht das erste Zeichen „c" im String „s" und liefert einen Zeiger darauf. Sollte in „s" kein „c" vorkommen, ist das Ergebnis Null.

strrchr

```
char *strrchr(const char *s, int c)
```

leifert analog zu „strchr" einen Zeiger auf das letzte „c" in „s" oder Null, falls es so etwas nicht gibt.

strspn

```
size_t strspn (const char *s, const char *c)
```

liefert die Anzahl der Zeichen am Anfang von „s", die alle auch in „c" vorkommen, z. B.

```
strspn("caddy", "abcd") == 4
```

strcspn

```
size_t strcspn(const char *s, const char *c)
```

ermittelt die Anzahl der Zeichen am Anfang von „s“, die allesamt nicht im String „c“ vorkommen.

strpbrk

```
char *strpbrk(const char *s, const char *c)
```

ist eine Art erweiterte Version von „strchr“ und liefert einen Zeiger auf die erste Stelle in „s“, an der irgendein Zeichen aus „c“ vorkommt.

strstr

```
char *strstr(const char *s1, const char *s2)
```

Diese Funktion sucht die komplette Zeichenkette „s2“ im String „s1“ und liefert in gewohnter Manier einen Zeiger auf ihr erstes Auftreten oder Null.

strlen

```
size_t strlen(const char *s)
```

liefert die Länge des Strings „s“, wobei das Nullzeichen am Ende nicht mitgerechnet wird.

strerror

```
char *strerror(int n)
```

liefert die Fehlermeldung zur Fehlernummer „n“, also im Wesentlichen das, was „perror“ ausgibt.

strtok

```
char *strtok(char *s, const char *c)
```

Mit der Funktion „strtok" kann man eine Zeichenkette „s" in eine Folge von Zeichenketten, die jeweils durch Zeichen aus „c" begrenzt sind, zerlegen. Beim ersten Aufruf ist für „s" ein Zeiger auf den Anfang der Zeichenkette zu setzen, bei allen weiteren ist hier „Null" zu setzen. Die Funktion sucht jeweils die erste Position im String, an der ein Zeichen aus „c" auftritt, setzt an diese Stelle ein ‚\0' und gibt einen Zeiger auf den Anfang dieses Teilstrings zurück. Dabei merkt sie sich die Position, an der das Zeichen aus „c" gefunden und das Nullzeichen gesetzt wurde, und sucht beim nächsten Aufruf, sofern als erstes Argument Null übergeben wird, von dieser Stelle weiter.

Wenn auf diese Weise der gesamte String verhackstückt wurde und kein Zeichen aus „c" mehr gefunden wurde, liefert „strtok" das Ergebnis Null.

Der sittliche Nährwert dieser Funktion liegt darin, eine Zeichenfolge „s" in eine Folge von Token, die durch Trennzeichen aus „c" separiert werden, zu zerbröseln. Beispielsweise zerlegt das folgende Programm einen String in Worte, wobei eine Zeichenfolge, die keine Leerzeichen oder Zeilentrenner enthält, als Wort betrachtet wird:

```
#include <string.h>
#include <stdio.h>

void main()
{
  char s[ ] = "Alles Seiende entsteht ohne Grund,\nsetzt
   sich aus Schwäche fort und stirbt durch Zufall.",
    trenn[ ] = " \n";
  char *pos;
  int i = 0;

  // erstes Wort abtrennen:
  pos = strtok(s,trenn);

  while(pos) // d. h, solange noch Worte gefunden werden:
  {
    // Wort ausgeben:
    printf("Wort %2i: %s\n", ++i, pos);

    // nächstes Wort ermitteln:
    pos = strtok(0,trenn);
  }
}
```

Das Programm liefert folgende Ausgabe:

```
Wort  1: Alles
Wort  2: Seiende
Wort  3: entsteht
Wort  4: ohne
Wort  5: Grund,
Wort  6: setzt
Wort  7: sich
Wort  8: aus
Wort  9: Schwäche
Wort 10: fort
Wort 11: und
Wort 12: stirbt
Wort 13: durch
Wort 14: Zufall.
```

memcpy

```
void *memcpy(void *v1, const void *v2, size_t n)
```

Von Adresse „v2" werden „n" Bytes an Adresse „v1" kopiert. Dabei dürfen sich die beiden Speicherbereiche nicht überlappen. Ergebnis der Funktion ist „v1".

memmove

```
void *memmove(void *v1, const void *v2, size_t n)
```

entspricht „memcpy", aber hier dürfen sich der Quell- und der Zielbereich auch überlappen.

memcmp

```
int memcmp(const void *v1, const void *v2, size_t n)
```

An den Adressen „v1" und „v2" werden die jeweils ersten „n" Zeichen miteinander verglichen. Diese Funktion ist identisch mit „strncpy", aber sie vergleicht beliebige Daten (allerdings immer byteweise) und bricht nicht beim ersten Nullbyte ab.

memchr

```
void *memchr(const void *v, int c, size_t n)
```

sucht analog zu „strchr“ das Zeichen „c“ in den ersten „n“ Zeichen des Speichervektor „v“ und liefert einen Zeiger auf dieses Zeichen.

memset

```
void *memset(void *v, int c, size_t n)
```

schreibt das Zeichen „c“ in die ersten „n“ Bytes ab Adresse „v“.

1.13 Datum und Uhrzeit: <time.h>

clock_t

```
typedef unsigned clock_t
```

Der ganzzahlige Datentyp „clock_t" repräsentiert eine Zeitangabe und wird im Zusammenhang mit der Funktion „clock" benutzt.

time_t

```
typedef unsigned time_t
```

Auch „time_t" ist ein Datentyp und stellt so etwas wie eine komprimierte Form der „DateStamp"-Struktur des Betriebssystems dar.

tm

```
struct tm
{ int tm_sec,  tm_min,  tm_hour, tm_mday, tm_mon, tm_year,
  tm_wday, tm_yday, tm_idst; }
```

Die Struktur „tm" enthält alles, was zu einer Zeit- und Datumsangabe gehört. Die einzelnen Felder bedeuten dabei:

tm_sec	Sekunden (0 bis 61, denn man hat hier lustigerweise sogar daran gedacht, daß es bis zu 2 Schaltsekunden geben kann)
tm_min	Minuten (0 bis 59)
tm_hour	Volle Stunden seit Mitternacht (0 bis 23)
tm_mday	Monatstag (1 bis 31)
tm_mon	Monate seit Januar (d. h. von 0 bis 11!)
tm_year	Jahre seit 1900, z. B1992
tm_wday	Wochentag: von 0 (Sonntag) bis 6 (Samstag)

tm_yday Tage seit dem 1. Januar (0 bis 365)

tm_idst positiv, wenn Sommerzeit gilt, Null bei Sommerzeit und negativ, wenn das unbekannt ist (in Maxon C++ immer)

time

```
time_t time (time_t *tp)
```

Die Funktion „time" liefert die aktuelle Systemzeit in Form einer „time_t"-Zahl. Wenn ihr Argument nicht Null ist, wird das Ergebnis zusätzlich noch in „*tp" abgelegt.

Normalerweise benutzt mandas Ergebnis von „time" mit den Funktionen „difftime", „gmtime" oder „localtime".

gmtime

```
struct tm *gmtime (const time_t *tp)
```

Der Inhalt von „*tp", also eine Zeitangabe in diesem geheimnisvollen „time_t"-Format, wird in einem ebenso geheimen, internen Puffer in das anwenderfreundliche „tm"-Format umgewandelt. Ein Zeiger auf diesen Puffer wird zurückgegeben, so daß man die Kalender- und Zeitdaten von dort auslesen kann. Es gibt aber nur einen internen Puffer, so daß die Daten beim nächsten Aufruf von „gmtime" oder ihrer Schwesterfunktion „localtime" wieder überschrieben werden.

localtime

```
struct tm *localtime(const time_t *tp)
```

Diese Funktion unterscheidet sich von „gmtime" nur darin, daß die Systemzeit in „*tp" hier in Ortszeit umgewandelt wird (daher das „local"), während „gmtime" ein Ergebnis in „Coordinated Universal Time" (UTC) liefern sollte - das „gm" leitet sich übrigens von der alten Bezeichnung „Greenwich Mean Time" ab. Unter Maxon C++ sind die beiden Funktionen aber absolut identisch.

mktime

```
time_t mktime (struct tm *tp)
```

„mktime" ist das Gegenstück zu „localtime" und wandelt den Inhalt der „tm"-Struktur „*tp" in das komprimierte „time_t"-Format um. Im Prinzip wird dabei verlangt, daß alle, aber auch wirklich alle Einträge der Struktur korrekt initialisiert sind. Maxon C++ ist aber schon zufrieden, wenn die ersten sechs Member (Sekunden, Minuten, Stunden, Tag, Monat, Jahr) mit gültigen Werten initialisiert sind.

clock

```
clock_t clock (void)
```

Diese Funktion liefert die bisherige Laufzeit des Programms. Maxon C++ implementiert diese Funktion etwas nachlässig, denn „clock" liefert nur die Zeit, die seit dem Programmstart vergangen ist, und nicht die CPU-Zeit, wie es eigentlich verlangt wird. Jedenfalls können Sie das Ergebnis von „clock" in Sekunden umrechnen, indem Sie durch „CLOCKS_PER_SEC" dividieren.

CLOCKS_PER_SEC

```
#define CLOCKS_PER_SEC 50
```

Diese Konstante gibt den Umrechnungsfaktor zwischen „clock_t" und einer Zeitangabe in Sekunden an. „clock()/CLOCKS_PER_SEC" liefert die bisherige Laufzeit des Programms in Sekunden.

difftime

```
double difftime(time_t t1, time_t t2)
```

„difftime" subtrahiert die Zeiten „t1-t2" und liefert das Ergebnis als Fließkommazahl in Sekunden.

strftime

```
int strftime(char *s, unsigned len, const char *format, const struct
tm *t)
```

Diese hochkomfortable Funktion wandelt die Zeit in der Struktur „*t"ähnlich wie die Funktion „sprintf" in eine Stringdarstellung um und legt diese mit einem abschließenden Nullzeichen im Zeichenvektor „s" ab. Dabei werden maximal „len" Zeichen in „s" geschrieben.

„format" ist ein Formatstring, der wie bei „sprintf" nach „s" kopiert wird. Dabei werden Anweisungsfolgen, die jeweils mit einem „%" beginnen, durch ansprechend formatierte aus „*t" ersetzt. Der Rückgabewert ist die Anzahl der Zeichen, die in „s" geschrieben wurden, wobei das abschließende Nullbyte nicht mitgezählt wird. Wenn aber „len" Zeichen nicht reichen, ist das Ergebnis Null.

Die folgende Tabelle gibt die zahlreichen möglichen Umwandlungsoperatoren an:

"	Gekürzter Wochentag, z. B. „Mon" für Montag
%A	Vollständiger Wochentag, z. B. „Monday"
%b	Monats-Kurzname, z. B. „Apr" füt April
%B	Voller Name des Monats, z. B. „April"
%c	Kurzdarstellung von Datum und Uhrzeit, z. B. „Apr 06 23:55:42 1992"
%d	Monatstag („01" bis „31")
%H	Amerikanische Stundendarstellung („01" bis „12")
%I	Stunde auf Europäisch („00" bis „23")
%j	Tag im Jahr („001" - „366")
%m	Monat („01" - „12")
%M	Minute („00" - „59")
%p	„AM" oder „PM"
%S	Sekunde („00" - „61")
%U	Woche im Jahr („00"-"53"), mit Sonntag als ersten Wochentag
%w	Wochentag (von „0" für Sonntag bis „6" für Samstag)
%W	Woche im Jahr („00"-"53"), mit Montag als ersten Wochentag
%x	Datum, z. B. „Apr 06 1992"
%X	Uhrzeit, z. B. „23:55:42"
%y	Jahr im Jahrhundert, z. B. „92"
%Y	Vollständige Jahreszahl, z. B. „1992"
%Z	Name der Zeitzone (hier nicht implementiert)
%%	Prozentzeichen

Beispiel:

```
#include <stream.h>
#include <time.h>

void main()
{
  char s[80];
  time_t t = time(0);

  strftime(s, 80, "Es ist jetzt %X am %d. %m. %Y.\n",
  localtime(&t));
  cout « s;
}
```

gibt so etwas aus wie

```
Es ist jetzt 10:47:11 am 07. 04. 1992.
```

asctime

```
char *asctime (const struct tm *t)
```

ist eine Sparversion von „strftime" und wandelt die Zeit aus „*t" in eine Zeichenkette der Form „Tue Apr 07 01:03:42 1992" um. Diese Zeichenkette wird in einem internen Puffer abgelegt und ein Zeiger darauf zurückgegeben.

ctime

```
char *ctime(const time_t *t)
```

ist identisch mit „asctime(localtime(t))", wandelt also einen „time_t"-Wert in eine Stringdarstellung um.

2. Bibliotheken für C++

2.1 Ein- und Ausgaben in C++: <stream.h> und <iostream.h>

2.1.1 Ausgabe

2.1.1.1 „ostream" und der Operator „«"

Bekanntlich ist in C++ alles viel schöner, besser und leichter als in ANSI C, und so ist es auch bei der Ein- und Ausgabe von Daten. Die dazu nötigen Definitionen stehen in der Includedatei „<stream.h>", während „<iostream.h>" nur so etwas wie ein Aliasname für erstere Datei ist und lediglich ein „#include <stream.h>" enthält. Genaugenommen ist „<stream.h>" die inzwischen veraltete Bezeichnung (aus dem 1.0-Standard) und „<iostream.h>" laut 2.0-Standard korrekter, aber letzten Endes ist das ja egal, denn die beiden Bezeichnungen sind absolut äquivalent. In diesem Handbuch wird konsequent „<stream.h>" benutzt.

In diesen Dateien wird unter anderem die Klasse „ostream" definiert, die für Ausgaben aller Art zuständig ist. Es gibt drei Standard-Objekte dieses Typs:

cout	für gewöhnliche Ausgaben
cerr	für Fehlermeldungen, die sofort ausgegeben werden
clog	für gepufferte Fehlermeldungen

Maxon C++ nimmt sich die Freiheit, daß „cerr" und „clog" unterschiedliche Namen für ein und dasselbe Objekt sind.

Auf dieser Klasse „ostream" ist vor allem der Operator „«" mehrfach überladen. Der linke Operand ist dabei jeweils ein „ostream"-Objekt, über das die Ausgabe erfolgen soll, z. B. „cout", und der rechte Operand ist ein Ausdruck eines Standard-Typs. Im einzelnen sind folgende Operatorfunktionen als Member definiert:

ostream &operator « (int)
ostream &operator « (unsigned)
ostream &operator « (long)
ostream &operator « (unsigned long)
ostream &operator « (long long)
ostream &operator « (unsigned long long)
ostream &operator « (unsigned char)
ostream &operator « (signed char)
ostream &operator « (char)

ostream &operator « (const char *)
ostream &operator « (float)
ostream &operator « (double)
ostream &operator « (long double)
ostream &operator « (const void*)
ostream &operator « (form &)
ostream &operator « (ios&(*f)(ios&))

Wie unschwer zu erkennen ist, gibt es für jeden numerischen Datentyp eine Ausgabefunktion (die Typen „short" und „unsigned short" werden gemäß den C++-Typregeln nach „int" umgewandelt und mit der entsprechenden Funktion ausgegeben). „signed char"- und „unsigned char"-Werte werden hier als Zahlen interpretiert, während bei „char" ein Zeichen ausgegeben wird. Bei einem Zeichenkettenargument („const char*") wird natürlich der entsprechende String ausgegeben, während bei einem beliebigen anderen Pointer („const void*") die Speicheradresse als hexadezimale Zahl mit vorangestelltem „0x" ausgegeben wird. Das kann natürlich zu Überraschungen führen, wenn man die Adresse einer „char"-Variablen ausgeben will:

```
#include <stream.h>

void main()
{
int i;
char c;

cout « &i;        // gibt Adresse von "i" hexadezimal aus
cout « &c;        // verunglückte String-Ausgabe
}
```

Hier versucht der Compiler, der nun einmal hinreichend dämlich ist, „c" als String zu interpretieren, und das Programm gibt alle Zeichen von „c" bis zum ersten im Speicher folgenden Nullbyte aus.

Außerdem stoßen die beiden letzten Funktionsdeklarationen in der obigen Liste bei Ihnen vielleicht auf Unverständnis. Die Funktion

```
ostream &ostream::operator « (form &)
```

stellt über die Klasse „form" eine formatierte Ausgabe im C-Stil zur Verfügung (siehe auch 2.1.1.3), und

```
ostream &ostream::operator « (ios&(*f)(ios&))
```

wendet eine Funktion auf eine Basisklasse von „ostream" an (2.1.1.3).

Das Ergebnis all dieser Funktionen ist stets eine Referenz auf das „ostream"-Objekt, so daß man mehrere Ausgaben aneinander hängen kann, z. B.:

```
int i=0x2a;
cout « "Der Wert von i ist: " « i « '\n';
```

Auch wenn die Werte natürlich in der Reihenfolge ausgegeben werden, in der man sie hinschreibt, kann ihre Auswertungsreihenfolge durchaus eine andere sein:

```
int i = 40;

cout « ++i « ++i;
```

Die Ausgabe kann hier entweder „4142" oder „4241" sein (die Ausgabe von Zahlen erfolgt ohne trennende oder führende Leerzeichen), denn voll ausgeschrieben entspricht dieser Ausdruck

```
cout.operator«(++i).operator«(++i)
```

und die Auswertungsreihenfolge in Ausdrücken ist in C++ nur in bestimmten Ausnahmefällen vorgeschrieben.

2.1.1.2 Basen und Manipulatoren

Die Klasse „ostream" hat noch eine Basisklasse namens „ios". Aus bestimmten Gründen wird „ostream" sogar virtuell davon abgeleitet, aber das interessiert momentan nicht so unbedingt. Jedenfalls existieren ein paar Funktionen, nämlich „dec", „hex", „oct", „endl" und „flush", die als Parameter eine Referenz auf ein solches „ios"-Objekt erwarten und dieselbe Referenz als Ergebnis liefern. Da eine abgeleitete Klasse alle Eigenschaften ihrer Basisklassen erbt, kann man diese Funktionen natürlich auch auf die Klasse „ostream" anwenden.

Die drei Funktionen

```
ios &dec (ios &s)
ios &hex (ios &s)
ios &oct (ios &s)
```

setzen die Zahlbasis in einem Strom-Objekt. Nach einem Funktionsaufruf wie

```
hex(cout)
```

werden in Zukunft über „cout“ alle ganzzahligen Werte hexadezimal, aber ohne vorangestelltes „0x“, ausgegeben. Die Ausgabe von Fließkomma- oder Stringwerten wird dadurch nicht beeinflußt.

Analog schaltet die Funktion „oct“ auf oktale Zahldarstellung um, und „dec“ stellt wieder den dezimalen Modus her.

Ein Beispiel:

```
#include <stream.h>

void main()
{
  int i = 42;
  hex(cout);
  cout « "hexadezimal: " « i « "\n";
  dec(cout);
  cout « "dezimal:     " « i « "\n";
  oct(cout);
  cout « "oktal:       " « i « "\n";
  cout « "Es gilt die jeweils zuletzt gewählte Einstellung: "
  « i « "\n";
}
```

Das Programm gibt aus:

```
hexadezimal: 2a
dezimal:     42
oktal:       52
Es gilt die jeweils zuletzt gewählte Einstellung: 52
```

Wie Sie an der letzten Ausgabe erkennen können, gelten die so vorgenommenen Einstellungen auch über eine Ausgabezeile hinaus, nämlich so lange, bis wieder ein anderer Modus gewählt wird.

Ein- und Ausgabedateien, und ein C++-Strom repräsentiert im Grunde genommen nichts anderes, werden oft mit einem Puffer beschleunigt (siehe auch 1.2.6.2). Die Funktion

```
ios &flush (ios &s)
```

löscht den Schreibpuffer eines Ausgabestroms, indem sie seinen Inhalt ausgibt. Wenn der Strom keinen Puffer hat, hat die Funktion keinerlei Wirkung. Maxon C++ weist den Standard-Ausgabeströmen als Default zwar keinen Puffer zu, aber auf einigen Systemen ist nach einer Ausgabe ein „flush(cout)“ bzw. „flush(clog)“ nötig, um die Ausgabe wirklich zu schreiben.

Natürlich kann man diese Funktion auch so benutzen:

```
flush(cout « "Ausgabe");
```

...oder so:

```
flush(cout) « "Ausgabe";
```

Der erste Ausdruck gibt erst etwas aus und wendet dann „flush" an, die zweite Anweisung schreibt erst einen etwaigen Pufferinhalt und macht dann eine weitere Ausgabe, die aber gegebenenfalls natürlich zunächst in einem Puffer aufbewahrt wird.

Last not least gibt es noch die Funktion

```
ios &endl (ios &s)
```

Sie gibt einen Zeilentrenner in einen Strom aus und ruft anschließend „flush" auf:

```
cout « "Das Ergebnis ist: " « Resultat;
endl(cout);
```

oder

```
endl(cout « "Das Ergebnis ist: " « Resultat);
```

entspricht

```
cout « "Das Ergebnis ist: " « Resultat;
flush(cout « "\n");
```

Diese kleinen Funktionen sind zwar ganz praktisch, aber ihre Handhabung und der damit verbundene Schreibaufwand entsprechen, wie auch die Übersichtlichkeit, doch eher dem C-Niveau als einem angenehmen C++-Stil. Deshalb gibt es die Funktion

```
ostream &ostream::operator « (ios&(*f)(ios&))
```

Wenn man die Adresse einer Funktion mit der passenden Signatur in ein „ostream"-Objekt „schreibt", wird sie darauf ausgeführt:

```
cerr « hex
```

ist damit äquivalent zu

```
hex(cerr)
```

Das eröffnet dem Programmierer reizvolle Möglichkeiten:

```
cout « hex « 26731 « endl;
```

schaltet die Standard-Ausgabe auf hexadezimale Ausgabe um, schreibt dann in eben diesem Modus eine Zahl und führt „endl" aus. Auf diese Weise wird die Handhabung solcher Funktionen doch erheblich angenehmer und intuitiver.

2.1.1.3 Formatierte Ausgabe

Die Klasse „form" stellt eine formatierte Ausgabe im Stil von „printf" zur Verfügung. Sie hat einen Konstruktor, dessen Parameter demgemäß eine nicht ganz unzufällige Ähnlichkeit mit denen von „printf" haben:

```
form(const char *fmt, ...)
```

In der Formatzeichenkette können sämtliche Umwandlungsoperationen und Optionen von „printf" (siehe 1.2.4) benutzt werden, und an Stelle der Ellipse „..." sind die entsprechenden zusätzlichen Argumente zu setzen. Man kann sich ein solches „form"-Objekt erzeugen und mit dem gewöhnlichen Operator „«" in einen „ostream" ausgeben:

```
#include <stream.h>

void main()
{
int i=42;
char s[ ] = "die große Frage";

form f("Die Antwort auf %s ist %d.\n", s, i);

cout « f;
}
```

In dem Objekt „f" wird hier der String

```
"Die Antwort auf die große Frage ist 42.\n"
```

abgespeichert und kann beliebig oft ausgegeben werden. Gebräuchlicher ist aber wohl die Möglichkeit, ein namenloses, temporäres Objekt auszugeben:

```
cout « form("Hex: %x", 42) « endl;
```

Auf diese Weise stehen auch bei den C++-Streams sämtliche Formatierungsmöglichkeiten

von „printf“ zur Verfügung. Der Nachteil ist natürlich, daß auch „form“ eine Funktion ohne Netz und doppelten Boden ist und der Programmierer deshalb selbst darauf zu achten hat, daß die Argumente stimmen.

2.1.1.4 Ausgabe binärer Daten

Die Standardausgabe ist in der Regel ein Fenster, oder sie wird auch manchmal in eine (Text-)Datei umgelenkt. Deshalb wird man normalerweise über „cout“, „cerr“ und „clog“ nur formatierte Textdaten ausgeben. Man kann einen „ostream“ aber auch an eine Datei anbinden, und dann wird man durchaus manchmal auch binäre Daten schreiben wollen. Deshalb gibt es die beiden Memberfunktionen

```
ostream &ostream::put (char c)
ostream &ostream::write (const char *p, int n)
```

Erstere Funktion schreibt ein Zeichen in einen Ausgabestrom, letztere die ersten „n“ Bytes ab Adresse „p“. Daß „p“ hier ein Zeiger auf „char“ ist, sollte Sie nicht weiter beunruhigen, denn natürlich können Sie beliebige andere Zeiger nach „char*“ casten.

Einige Beispiele:

```
cout.write("Hello, World!",5)
```

schreibt die Zeichenfolge „Hello“ nach „cout“, und

```
double d = 3.14159;
cout.write((char*) &d, sizeof(double));
```

schreibt die „double“-Zahl „d“ in denselben Strom, aber nicht als lesbare, formatierte Zahldarstellung, sondern in Form von (hier)acht Bytes.

Das erste Beispiel ist durchaus sinnvoll (manchmal möchte man wirklich nur einen Teil eines Strings ausgeben, oder man hat eine Zeichenkette vorliegen, bei der das Nullbyte am Ende fehlt), während das letztere wirklich nur bei Dateioperationen einen sittlichen Nährwert hat.

2.1.2 Eingabe

2.1.2.1 Texteingaben mit „»“

Was Krupp in Essen, wir im Trinken und „ostream“ bei der Ausgabe, ist die Klasse „istream“ für die Eingabe von Daten. Es gibt ein Standard-Objekt dieser Klasse namens „cin“, das in „<stream.h>“ deklariert wird und die Standard-Eingabe eines Programms darstellt, also normalerweise die Eingaben des Benutzers.

Die folgenden Operatoren sind als Member von „istream“ deklariert:

istream & operator » (char s[ ])
istream & operator » (char &c)
istream & operator » (signed char &sc)
istream & operator » (unsigned char &uc)
istream & operator » (short &s)
istream & operator » (unsigned short &us)
istream & operator » (int &i)
istream & operator » (unsigned &u)
istream & operator » (long &l)
istream & operator » (unsigned long &ul)
istream & operator » (long long &vl)
istream & operator » (unsigned long long &uvl)
istream & operator » (float &f)
istream & operator » (double &d)
istream & operator » (long double &ld)

Linker Operand ist jeweils ein Objekt der Klasse „istream“, also insbesondere die Variable „cin“, und eine Referenz auf genau dieses Objekt ist auch der Rückgabewert dieser Operatorfunktionen. Als rechter Operand kann, wie Sie der imposanten Auflistung von Funktionen unschwer entnehmen können, ein L-Wert eines beliebigen numerischen Standardtyps (einschließlich „char“) oder eine Stringvariable dienen. Die Operatorfunktion liest dann die benötigte Anzahl von Zeichen aus dem Eingabestrom, wandelt sie in den gewünschten Datentypen um (z. B. eine Ziffernfolge in eine ganze Zahl) und weist das Resultat dem rechten Operanden zu.

Ein Beispiel:

```
int i;
char c;
char s[50];

cin » c » i » s;
```

Bei der Eingabe

```
0815-4711 42
```

wird „c" das Zeichen „0" zugewiesen, „i" bekommt den Wert „815" und in „s" wird die Zeichenkette „-4711" abgelegt, während die Zeichen „42" vorerst unbeachtet bleiben und vielleicht bei der nächsten Eingabe Beachtung finden. Liest man auf diese Weise ein „char", werden zuerst Leerzeichen, Zeilentrenner und sonstige Leerraum-Zeichen überlesen. Beim Lesen von Strings werden ebenfalls Trennzeichen überlesen, und es wird bis ausschließlich zum nächsten nachfolgenden Trennzeichen gelesen. Somit wirkt also - zumindest beim Operator „»" - auch ein Leerzeichen als Trennung zwischen zwei Strings.

Auch die Klasse „istream" kann nicht hellsehen, und deshalb weiß die Eingabe-Operatorfunktion „»", wenn sie auf eine Stringvariable angewandt wird, nicht, wie lang dieser String ist. Das kann zu üblen Abstürzen führen, wenn der Anwender bei

```
char input[20];

cin » input;
```

mehr als 20 Zeichen eingibt. Um dies halbwegs sicher zu gestalten, liest die Funktion in Maxon C++ maximal „STREAM_MAXSTRING" Zeichen in Stringvariablen, und zwar einschließlich des Nullbytes am Ende. Diese Konstante wird in „<stream.h>" als 80 definiert - ein kanonischer Wert für eine vernünftige Zeilenlänge. Leider handelt es sich dabei um eine Spezialität von Maxon C++ und nicht um einen wirklichen Standard.

Ganzzahlige Werte können wahlweise dezimal, oktal (mit führender Null) oder hexadezimal (mit „0x" oder „0X" am Anfang) eingegeben werden. Fließkommazahlen werden natürlich nur dezimal akzeptiert und können einen Exponententeil (mit „e" oder „E" eingeleitet) haben.

2.1.2.2 Eingabefunktionen auf „istream"

Die Memberfunktion „read" liest aus einem „istream" eine beliebige Anzahl von Bytes und legt sie in einem Zielvektor ab. Damit ist sie das Gegenstück zur ANSI-Funktion „fread". Sie ist definiert als

```
istream & istream::read(char buf, int n)
```

Auch hier sollte es Sie nicht weiter beunruhigen, daß der Zielvektor „buf" formal ein „char"-Array ist, denn man kann natürlich jeden Pointer in ein „char*" umwandeln. „n"ist die Anzahl der Bytes, die gelesen werden sollen, und muß natürlich positiv sein. Das Ergebnis der Funktion ist wie immer eine Referenz auf das „istream"-Objekt.

Drei weitere Funktionen, die allesamt „get“ heißen, lesen jeweils ein Zeichen aus dem Eingabestrom und sind folgendermaßen definiert:

```
istream &  istream::get (char &c)
istream &  istream::get (unsigned char &c)
int        istream::get ()
```

Die beiden ersten Funktionen lesen das Zeichen in eine „char“-Variable und liefern die übliche Referenz als Ergebnis. Sie sind gleichermaßen auf „char“ und „unsigned char“ definiert. Die dritte Funktion expandiert das Zeichen nach „int“ und liefert es als Ergebnis.

Der Unterschied zwischen

```
cin » c
```

und

```
cin.get(c)
```

besteht darin, daß „get“ keinen Leerraum überspringt, also zeichenweise die vollständige Eingabe einschließlich aller Leerzeichen und Zeilentrenner liest.

Oft merkt man, daß man genug gelesen hat, erst, nachdem man schon zu viel gelesen hat. In diesem Fall hilft die Funktion „putback“, die ein bereits gelesenes Zeichen in den Eingabestrom zurückstellt:

```
istream & istream::putback(char c)
```

Auf diese Weise kann aber immer höchstens ein Zeichen gepuffert werden.

Besonders beim Lesen aus Dateien muß man irgendwie wissen, wann man am Ende der Datei angelangt ist. Deshalb bietet „istream“ auch eine Memberfunktion, die eben dieses testet:

```
int istream::eof()
```

Das Ergebnis dieser Funktion ist von Null verschieden, wenn das Dateiende überschritten wurde. Damit entspricht die Funktion „feof“ aus „<stdio.h>“.

Nun fehlt noch eine Funktion, die eine Textzeile liest. Hier ist sie:

```
istream &getline(char *buf, int len, char Delim = '\n')
```

Es werden maximal „len"-1 Zeichen in den Vektor „buf" gelesen. Die Funktion bricht ab, sobald ein Trennzeichen „Delim" gelesen wird - als Default ist dies der Zeilentrenner. Dieses Trennzeichen wird zwar aus dem Eingabestrom gelesen und nicht wieder zurückgestellt, aber nicht in „buf" abgelegt.

2.1.3 Weitere Funktionen

2.1.3.1 Die Schnittstelle zum ANSI C-Dateisystem

Oben wurde bereits angedeutet, daß die „stream"-Funktionen von Maxon C++ im wesentlichen verkleidete ANSI-C-Funktionen sind. Diese Schnittstelle kann der Programmierer durch die Funktion „getstream" der Klasse „ios" nutzen:

```
FILE * ios::getstream()
```

Diese Funktion liefert als Ergebnis einen Zeiger auf den ANSI C- Filedeskriptor. Damit lassen sich sämtliche Funktionen aus „<stdio.h>" auch auf C++-Streams benutzen.

Außer den hinlänglich bekannten Objekten „cout", „cin" und so weiter kann man auch selbst Objekte der Klassen „istream" und „ostream" erzeugen. Zu diesem Zweck haben die beiden genannten Klassen jeweils einen Konstruktor, der als Argument einen Dateizeiger erwartet.
Ein Beispiel:

```
#include <stream.h>
#include <stdio.h>

void main()
{
  FILE *f = fopen("test", "r");// C-File öffnen
  if (f)
  {
    istream is(f);              // eigenes "istream"-Objekt
    char s[80];

    is » s;                     // liest einen String

    fclose(f);                  // Datei wieder schließen
  }
}
```

„istream" und „ostream" haben keine Destruktoren, die die Dateien nachher automatisch schließen. Schon allein deshalb ist es nicht unbedingt sinnvoll, auf diese Weise mit Dateien zu arbeiten - mit den Klassen und Funktionen aus „<fstream.h>" geht das viel einfacher.

2.1.3.2 Fehlererkennung

Mit einer etwas eigenartigen Funktion bieten die Strom-Klassen eine äußerst intuitive Möglichkeit, Fehler in Dateien zu erkennen: In der Basisklasse „ios" gibt es eine Konvertierungsfunktion

```
ios::operator void*()
```

Diese Funktion liefert „0", wenn das Dateiende überschritten oder ein anderer Fehler bemerkt wurde, und andernfalls einen Zeiger auf das Objekt selbst. Das sieht zunächst seltsam aus, ist aber um so einfacher anzuwenden: Damit kann man Objekte und Ausdrücke des Typs „ios" oder davon abgeleiteter Klassen direkt in Abfragen einsetzen, z. B. so:

```
cin » s;
if (cin)
{ usw.
```

...oder auch so:

```
char s[80];

while (cin » s)
cout « s;
```

Solche Ausdrücke sind logisch wahr, wenn kein Fehler aufgetreten und das Dateiende nicht erreicht ist. Natürlich kann man auf solche Abfragen auch die logischen Operatoren „!", „&&" und „||" anwenden.

Kanonisch wäre natürlich gewesen, statt dieses komischen „void"-Zeigers eine Konvertierungsfunktion nach „int" oder einem anderen numerischen Typ zu implementieren, denn ein Zeiger als Bedingung ist doch ein wenig gewöhnungsbedürftig. Das ist aber nicht möglich, weil die Standard-Operatoren „»" und „«" auf ganzzahligen Werten definiert sind und Ausdrücke wie „cout « n" damit mehrdeutig würden.

2.2 Ströme und Dateien: <fstream.h>

2.2.1 Klassen für Dateien

Die Standard-Ein- und Ausgabeströme sind zwar ganz nett, aber natürlich braucht man daneben auch „richtige" Dateien, und es wäre angenehm, wenn man alle Features aus „<stream.h>" auch mit beliebigen Dateien benutzen könnte. Deshalb gibt es die Includedatei „<fstream.h>", in der Klassen und Funktionen für Dateien definiert werden.

In <fstream.h> werden drei für Sie interessante Klassen definiert:

- „ifstream" für Eingaben aus Dateien,
- „ofstream" für Ausgaben in Dateien,
- „fstream" für Ein- und Ausgaben.

„ifstream" ist von „istream" (und damit indirekt von „ios") abgeleitet und erbt damit alle in 2.1.2 vorgestellten Eigenschaften. Entsprechend wurde „ofstream" von „ostream" abgeleitet, während „fstream" - ein interessantes Beispiel für Mehrfachvererbung - gleichzeitig „istream" und „ostream" als Basisklassen hat.

Die drei neuen Klassen haben außer diesen ererbten Funktionen auch noch einen Satz von elementaren Dateifunktionen, etwa Öffnen und Schließen von Dateien. Diese gemeinsamen Eigenschaften werden in der Klasse „fstreambase" zusammengefaßt, die ebenfalls Basisklasse von „ifstream", „ofstream" und „fstream" ist.

2.2.2 Dateien öffnen und schließen

Die drei oben vorgestellten Datei-Stromklassen besitzen jeweils einen Default-Konstruktor. Wird aber ein Klassenobjekt damit erzeugt, ist es zunächst an keine physikalische Datei „angebunden", und alle Ein- und Ausgabeoperationen führen unabwendbar zu einem Absturz. Deshalb muß zunächst die Member-Funktion „open" der gemeinsamen Basisklasse „fstreambase" aufgerufen werden:

```
FILE * fstreambase::open (const char *name, int mode)
```

Der erste Parameter steht, wie man sich denken kann, für den Dateinamen, während der zweite den Zugriffsmodus festlegt. Mögliche Werte dafür werden von einem Aufzählungstyp, der in der Klasse „ios" verborgen ist, definiert:

```
enum open_mode
{
in  = 1,          // Eingaben
out = 2,          // Ausgaben
app = 4           // Anhängen
}
```

Beim Modus „out" wird stets eine neue Datei erzeugt. Existiert bereits eine Datei gleichen Namens, so wird diese gelöscht.

Das folgende Programm öffnet eine Datei und schreibt eine Zeile hinein:

```
#include <fstream.h>

const char dateiname[ ] = "ram:Test";

void main()
{
ofstream os;
os.open(dateiname, ios::out);
os « "Dies ist die Datei " « dateiname « ".\n";
os.close();
}
```

Wie Sie sehen, haben wir oben bereits eine weitere Funktion aus der Klasse „fstreambase" benutzt, nämlich „close". Diese Funktion schließt die Datei, an die der Strom gebunden ist. Ein überflüssiges „close" schadet nicht. Nach dem Schließen ist das Stream-Objekt wieder an keinerlei Datei angekoppelt, und man kann, wenn man will, erneut eine Datei öffnen.

Kommen wir noch einmal auf die Zugriffsmodi von „open" zurück: Die drei vorgegebenen Werte können durch bitweises „Oder" miteinander kombiniert werden. Beispielsweise steht

```
ios::in|ios::out
```

für Lesen und Schreiben zugleich, oder beim Modus

```
ios::out|ios::app
```

wird zwar aus der Datei gelesen, aber der Lesezeiger steht zunächst am Ende der Datei und muß dann z. B. mit „seekg" (siehe unten) zurückgesetzt werden.

2.2.3 Fehlererkennung

Selbstverständlich sollte man immer, wenn man eine Datei öffnen läßt, prüfen, ob das auch wirklich geklappt hat, bevor man etwas mit der Datei anstellt. Dazu dient wieder die normale Fehlerkontrolle mit der eigenartigen Konvertierungsfunktion aus 2.1.3.2. Diese ist nämlich immer logisch „falsch", wenn ein Strom an keine Datei angekoppelt ist.

Das folgende Programm demonstriert dies, indem es eine Datei zum Lesen öffnet und sie dann in eine andere Datei kopiert:

```
#include <fstream.h>

void main()
{
  ifstream is;
  is.open("s:startup-sequence", ios::in);

  if(is)
  {
    ofstream os;
    os.open("ram:copy_of_s-seq", ios::out);

    if (os)
    {
      char s[80];
      while (is.getline(s,80))
        os « s « "\n";
      os.close();
    }
    else
      cout « "Kann Ausgabedatei nicht öffnen";

    is.close();
  }
  else
    cout « "Kann Eingabedatei nicht öffnen";
}
```

2.2.4 Konstruktoren und Destruktoren

Natürlich geht in C++ alles viel einfacher als in C, und so ist es auch bei diesen Dateioperationen. Ich verschwieg Ihnen bisher, daß „ifstream", „ofstream" und „fstream" jeweils nicht nur einen Default-Konstruktor haben, sondern auch einen solchen, dessen Parameter mit denen der „open"-Funktion identisch sind. Dadurch kann man die Definition eines Dateiobjekts mit dem Öffnen der Datei aufs einfachste kombinieren, z. B. so:

```
ofstream outf("C++:Ausgabe", ios::out);
```

Außerdem ist für diesen Konstruktor bei „ofstream" als Default-Argument für den zweiten Parameter „ios::out" definiert, so daß die Deklaration sich verkürzt zu:

```
ofstream outf("C++:Ausgabe");
```

Einen entsprechenden Konstruktor gibt es auch für die Klasse „ifstream", nur natürlich mit „ios::in" als Default-Argument. Auch für „fstream" existiert ein solcher Konstruktor, hier aber ohne einen abkürzenden Default-Modus.

Auch nachdem man ein Objekt auf diese erzeugt hat, muß man natürlich zuerst prüfen, ob die Datei auch wirklich geöffnet werden konnte, bevor man irgendwelche Ein- oder Ausgaben tätigt.

Außerdem haben die drei File-Stream-Klassen jeweils einen Destruktor, der die an das Objekt angebundene Datei schließt, sofern dies nicht schon geschehen ist. Er macht das natürlich unabhängig davon, mit welchem Konstruktor das Objekt erzeugt wurde.

Als Beispiel folgt noch einmal das in 2.2.3 vorgestellte Programm, diesmal aber unter Ausnutzung der Kon- und Destruktoren und mit vereinfachter Fehlerbehandlung:

```
#include <fstream.h>

void main()
{ ifstream is("s:startup-sequence", ios::in);
  ofstream os("ram:another_copy_of_s-seq", ios::out);

  if(is && os)
  { char s[80];
    while (is.getline(s,80))
      os « s « "\n";
    os.close();
  }
  else
    cout « "Kann nicht beide Dateien öffnen.";
}
```

2.2.5 Puffer

Auch bei den Strömen gibt es wieder das alte Elend, daß Dateioperationen nervtötend langsam sind, wenn byteweise oder in ähnlich kleinen Happen gelesen oder geschrieben werden muß. Deshalb gibt es in der Klasse „fstreambase" (also der gemeinsamen Basisklasse der drei Stromklassen) zwei Funktionen, deren Funktion sich dem geneigten Leser möglicherweise intuitiv erschließt:

```
int fstreambase::buffer (unsigned s)
void fstreambase::flush ()
```

„buffer" verleiht einer Datei einen Puffer der Größe „s" und entspricht damit einem „<stdio-.h>"-Aufruf wie

```
setvbuf(getstream(), 0, _IOFBF, s)
```

Man kann einen Puffer für eine Datei immer nur einmal setzen und danach seine Größe nicht mehr verändern. Der benötigte Speicherplatz wird automatisch angefordert und beim Schließen der Datei ebenso automatisch wieder freigegeben. Das Ergebnis von „buffer" ist übrigens Null, wenn dabei kein Fehler auftrat.

„flush" schreibt den Inhalt eines Schreibpuffers in die Datei.

2.2.6 Positionieren in Strömen

Von den grundlegenden Dateioperationen fehlen jetzt nur noch Operationen zum Ermitteln und Setzen der Dateiposition. Zunächst wäre da die Funktion

```
int fstreambase::seekg (long offset, int mode = ios::beg)
```

Hier gibt es drei verschiedene Modi, die in Form eines Aufzählungstyps versteckt in „ios" definiert werden:

```
enum seek_dir
{
  beg = -1, // absolute Position zum Dateianfang
  cur = 0,  // relative Verschiebung von der akt. Position
  end = 1   // (negativer) Offset zum Dateiende
}
```

Das folgende Programm macht dem Rechner ein „x" für ein „u" vor, indem es in einer Datei alle „u" und „U" durch „x" bzw. „X" ersetzt. Dazu braucht man natürlich ein

„fstream"-Objekt, denn man muß ja gleichermaßen lesen (um die „U"'s zu suchen) und schreiben (nämlich die „X"-e). Die Datei wird hier zeichenweise eingelesen, und immer, wenn ein „U" gefunden wurde, wird die Position mit „seekg" um ein Zeichen zurückgesetzt und dann eben dieses „U" überschrieben:

```
#include <fstream.h>

const char name[ ] = "ram:test";

void main()
{
  fstream f(name, ios::in|ios::out);

  while (f)
  {
    char c = f.get();                  // Zeichen lesen

    if (c == 'u' || c == 'U')          // "U" ?
    {
      f.seekg(-1, ios::cur);           // dann zurück
      f.put( c=='u' ? 'x' : 'X' );     // und "X" schreiben
    }
  }
}
```

Als letztes lernen Sie jetzt noch eine Funktion kennen, mit der Sie die aktuelle Dateiposition ermitteln können. Sie heißt „tellg":

```
long fstreambase::tellg ()
```

und liefert, wenn sie auf einem „ifstream"-, „ofstream"- oder „fstream"-Objekt aufgerufen wird, die Position bezüglich des Dateianfangs in Bytes. Die erste Position hat wie immer die Nummer Null.

2.3 Ein paar wichtige Definitionen in <new.h>

In der Datei „<new.h>“ steht eigentlich fast nichts: Zum einen wäre da der Datentyp „size_t“, der mit „unsigned“ identisch ist und auch in diversen anderen Files definiert wird, außerdem die hinlänglich bekannte Konstante „NULL“ und die folgende Funktion:

```
void (*set_new_handler(void(*)(void))) (void)
```

Man muß natürlich dreimal hinsehen, um diese Deklaration als Funktion zu erkennen. Einfacher wird's, wenn man die Deklaration mit einem „typedef“ aufteilt:

```
typedef void (*ftype) (void);
ftype set_new_handler (ftype);
```

Also erwartet die Funktion „set_new_handler“ als Argument einen Zeiger auf eine Funktion, die weder Parameter noch Ergebnistyp hat, und gibt auch einen solchen zurück. Sinn und Zweck dieser Funktion ist es, eine Handler-Funktion zu setzen, die Speichermangel bei „new“ oder „malloc“ handhabt.

Normalerweise ignoriert es die Laufzeitbibliothek, wenn bei „new“ oder „malloc“ kein Speicher angefordert werden konnte, und liefert dann lediglich einen Nullzeiger, mit dem das Programm dann gefälligst selbst klarkommen muß. Rein praktisch bedeutet das, daß hinter jedes „new“, „malloc“, „calloc“ usw. so etwas wie eine „if“-Abfrage gehört, die eine solche fehlgeschlagene Speicheranforderung abfängt und das Programm dann irgendwie sinnvoll weiterführt oder sauber beendet.

Bequemer kann man dieses lösen, indem man mit „set_new_handler“ eine Funktion setzt, die in Zukunft immer dann aufgerufen wird, wenn der Speicher nicht ausreicht. Nach Ausführung dieser Funktion (sofern dort nicht mit „exit“ o. ä. ausgestiegen wird) versucht die Bibliothek noch einmal, den benötigten Speicher anzufordern, und ruft im Mißerfolgsfall erneut den Handler auf.

Wählt man einen Nullzeiger als Argument für „set_new_handler“, ist wieder die Standard-Einstellung aktiv, d. h. das System kümmert sich nicht weiter um den Speicherüberlauf. Das Ergebnis von „set_new_handler“ ist in jedem Fall die bisher gültige Einstellung, also ein Zeiger auf eine zuvor gesetzte Handlerfunktion oder eben Null.

Die Funktion „set_new_handler“ wird auch in Abschnitt 2.6.6 des C++-Tutorials erläutert. Das folgende Beispielprogramm fordert gnadenlos so lange Speicherblöcke an, bis der Handler „coredumped“ aufgerufen wird.

```
#include <new.h>
#include <stream.h>

unsigned MemUsed = 0;

void coredumped()
{
  cout « "Speicherüberlauf nach " « MemUsed « " Bytes.\n";
  cout « "Programm steigt aus!!!\n";
  exit(42);
}

void main()
{
  const unsigned int size = 10000;

  set_new_handler(coredumped);

  for(;;)
  {
    char *cp = new char[size];
    MemUsed += size;
  }
}
```

2.4 Eine String-Library: <tools/str.h>

2.4.1 Die Klasse „String"

Im Unterverzeichnis „tools", das für Hilfsbibliotheken aller Art vorgesehen ist, finden Sie eine Datei „str.h", die Ihnen eine C++ angemessene Stringverwaltung beschert.

Im wesentlichen wird dort eine Klasse namens „String" deklariert, die dynamische Zeichenketten repräsentiert. Das bedeutet: Man deklariert lediglich eine Variable des Typs „String", und die Bibliothek sorgt selbst dafür, daß stets genug - und, einmal abgsehen von einem gewissen Overhead, auch nie zuviel - Speicher dafür reserviert wird. Im Tutorial wird in Kapitel 4.2.3 eine ähnliche (aber weniger leistungsfähige) Library als Beispielprogramm vorgestellt und implementiert.

Die Klasse „String" besitzt vier verschiedene Konstruktoren:

```
String (const char *s = 0)
String (int l, char *s)
String (const String &S)
String (char c)
```

Der erste Konstruktor initialisiert einen String mit einer Zeichenkette (oder dem Leerstring als Default) und dürfte der bei Variablendeklarationen meistgebrauchte sein, z. B.

```
String s1, s2 = "Horrido!", s3 = 0, s4 = "";
```

Hier werden die Variablen „s1", „s3" und „s4" jeweils als Leerstrings initialisiert, während in „s2" die Zeichenkette „Horrido!" abgespeichert wird. Dazu muß der Konstruktor natürlich Speicher anfordern und dies in „s2" vermerken. Um solchen Speicher wieder freizugeben, wenn die Stringvariable nicht mehr existiert, besitzt die Klasse „String" auch einen Destruktor.

Den zweite Konstruktor benötigt man hauptsächlich, wenn man der Bibliothek eigene Stringfunktionen hinzufügen will. Aus einem „char[ ]"-Objekt, das mit „new" erzeugt worden sein muß und mit einem String, der ausschließlich des abschließenden Nullbytes die Länge „l" hat, initialisiert sein muß, wird hier ein „String"-Objekt erzeugt. Dabei wird die Zeichenkette „s" nicht noch einmal kopiert, sondern der Zeiger darauf direkt in das Stringobjekt eingetragen.

Der dritte Konstruktor ist, wie man sieht, ein Copy-Konstruktor, und der letzte initialisiert das Objekt mit einem Zeichen, das als Zeichenkette der Länge „1" interpretiert wird. Damit sind

```
String s1 = "?"
```

und

```
String s1 = '?'
```

mehr oder weniger äquivalent.

2.4.2 Member-Funktionen

Mit den oben aufgeführten Konstruktoren gibt es eine Konvertierung von den ordinären Zeichenvektoren, mit denen sich der C-Programmierer herumschlagen muß, in die Stringklasse. Für Kompatibilität in die andere Richtung sorgt die Konvertierungsfunktion

```
String::operator char*() const
```

Sie wandelt ein „String"-Objekt in einen Zeiger auf eine Zeichekette um. Dabei sollte man sich aber hüten, diese Konvertierung auf temporäre Objekte anzuwenden, z. B. auf das Ergebnis der unten vorgestellten Operatorfunktion „+", denn natürlich liefert sie lediglich einen Zeiger auf den ansonsten „geheimen" Speicherbereich, in dem der eigentliche Wert des Stringobjekts abgespeichert wird, und wenn die Stringvariable nicht mehr existiert oder einen neuen Wert zugewiesen bekommt, muß man stets damit rechnen, daß dieser Speicherbereich freigegeben und wieder überschrieben wird.

Es gibt zwei Zuweisungsoperatoren:

```
String & String::operator = (const String &S)
String & String::operator = (const char *s)
```

Die erste Funktion funktioniert, wie Sie sich denken können, analog zum Kopier-Konstruktor, gibt aber den Speicher, der für den bisherigen Variablenwert alloziert wurde, wieder frei. Die zweite ist eigentlich redundant, denn der Compiler ist natürlich durchaus clever genug, bei einer Wertzuweisung wie

```
String s1;
char c[ ] = "Frood";

s1 = c;
```

aus dem Zeichenvektor mit Hilfe des entsprechenden Konstruktors ein temporäres Objekt zu erzeugen und dieses dann mit der erstgenannten „="-Funktion in das Zielobjekt zu kopieren. Mit dieser zusätzlichen Funktion wird dies aber deutlich abgekürzt.

Ungefähr wie „strcat“ in Standard-C, nur eben erheblich bequemer, funktioniert die Member-Funktion

```
String & String::operator += (const String &s),
```

die zwei Strings aneinander hängt: Nach

```
String s1 = "Moin ", s2 = "Moin!";

s1 += s2;
```

hat „s1“ den Wert „Moin Moin!“.

Sie wollen auf einen String zeichenweise zugreifen? Null Problemo, auch das geht, und zwar mit der folgenden Operator-Funktion:

```
char String::operator[ ] (int i) const
```

Sie liefert als Ergebnis das „i“-te Zeichen einer Stringvariablen, allerdings nicht als Referenz und damit auch nicht als L-Wert. Deshalb kann man mit Indizierung Zeichen aus einem String auslesen, aber keine einzelnen Zeichen verändern. Dies wurde aus Sicherheitsgründen so eingerichtet.

Um die Länge eines Strings festzustellen, können Sie die Standard-Funktion „strlen“ benutzen, denn es existiert ja eine Konvertierung von „String“ nach „char*“. Angenehmer und schneller ist aber die folgende Member-Funktion:

```
int String::length() const
```

Sie liefert die Länge eines Strings, wobei das Nullbyte am Ende wie immer nicht mitgezählt wird. Aufgerufen wird sie natürlich in der Form

```
i = s1.length();
```

Zu guter Letzt wären da noch drei Funktionen, die irgendwie an alte BASIC-Tage erinnern:

```
String String::left (int n) const
String String::right (int n) const
String String::mid (int pos, int n) const
```

„left“ liefert die ersten „n“ Zeichen eines Strings, „right“ die letzten „n“. Falls der String kürzer als „n“ ist, ist das Ergebnis mit dem Argument identisch. „mid“ nimmt aus einem String „n“ Zeichen ab der Position „pos“, wobei letztere wie immer von Null an gezählt wird.

2.4.3 Der Operator „+“ auf Strings

Das waren alle Funktionen, die als Member von „String“ deklariert sind. Daneben gibt es noch einige global deklarierte Operatorfunktionen:

Die angenehmste ist sicherlich

```
String operator + (String s1, String s2)
```

Dieses „+“ hängt, wie man es auch nicht anders erwartet, zwei Strings aneinander. Anders als bei „strcat“ oder dem Operator „+=“ werden hier die Werte der Argumente keineswegs verändert, sondern es wird ein neues (temporäres) Objekt erzeugt, falls das Ergebnis nicht sogar direkt in das gewünschte Ziel geschrieben wird.

Da es sonst keine Addition auf Standard-Strings (d. h. char-Zeigern) gibt, kann dieser neue Plus-Operator auch dort angewendet werden:

```
#include <tools/str.h>
#include <stream.h>

void main()
{ char *s1 = "Grunz",
  *s2 = s1 + "wanzling";

  cout « s2;
}
```

Eigentlich ist dieses Beispiel korrekt, denn bei der Addition wird der „+“-Operator aus der String-Bibliothek benutzt und dessen Ergebnis, ein temporäres Objekt der Klasse „String“, mit der Konvertierungsfunktion nach „char*“ gewandelt. Dabei wird aber der alte Fehler begangen, einen Zeiger auf ein temporäres Objekt zu verwenden. Wenn „s2“ ausgegeben wird, existiert das temporäre Objekt (und mit ihm sein Zeichenvektor) schon lange nicht mehr, und die Ausgabe ist undefiniert.

So hätte es natürlich geklappt:

```
#include <tools/str.h>
#include <stream.h>

void main()
{ char *s1 = "Grunz";
  String s2 = s1 + "wanzling";

  cout « s2;    // Ausgabe für "String" existiert - siehe unten
}
```

Hier wird - je nach Compiler - die String-Summe entweder gleich im Zielobjekt „s2" konstruiert, oder das hier eingeführte temporäre Objekt wird mit dem Copy-Konstruktor in „s2" übertragen, bevor es destruiert wird. Also funktioniert „+" hier so, wie man es erwartet.

Deshalb soll hier noch einmal erwähnt werden, daß die Konvertierungsfunktion von „String" nach „char*" nur mit Vorsichtig und Bedacht zu benutzen ist, denn andernfalls erlebt man Überraschungen.

2.4.4 Vergleiche

Natürlich kann man String-Objekte mit der Funktion „strcmp" vergleichen. Mit den folgenden Operatoren geht das aber wesentlich bequemer:

```
int operator == (const String &s1, const String &s2)
int operator != (const String &s1, const String &s2)
int operator <  (const String &s1, const String &s2)
int operator >  (const String &s1, const String &s2)
int operator <= (const String &s1, const String &s2)
int operator >= (const String &s1, const String &s2)
```

Die beiden Operanden werden jeweils wie mit „strcmp" verglichen (ASCII-Reihenfolge), und das Ergebnis ist ein boolscher Wert.

Ein Mini-Beispiel:

```
#include <tools/str.h>
#include <stream.h>

void main()
{ String s1 = "Test", s2 = "Text";

  if (s1 < s2)
    cout « s1;
  else
    cout « s2;
}
```

Ein kleiner Wermutstropfen: Vergleiche wie

```
String s1;

if(s1 == "Nanu?") ...
```

sind nicht erlaubt, da mehrdeutig! Schließlich gibt es sämtliche Vergleichsoperationen auch auf Zeigern (und insbesondere Zeichenketten), und so weiß der Compiler oben nicht, ob er „s1" nach „char*" konvertieren und diese Adresse dann mit der des Strings „Nanu?" vergleichen, oder ob er aus der konstanten Zeichenkette mittels Konstruktoraufruf ein temporäres „String"-Objekt erzeugen und darauf den String-Operator „==" anwenden soll.

So geht's aber:

```
if (s1 == String("Nanu?"))
```

2.4.5 Ein- und Ausgabe

In den obigen Beispielen wurden mehrfach Objekte der Klasse „String" ausgegeben. Das ist ohne weiteres möglich, denn die Operatoren wurden - wie im Tutorial unter Abschnitt 4.2.2.7 beschrieben - entsprechend zusätzlich überladen:

```
class ostream& operator « (ostream&, const String&)

class istream& operator » (istream&, String&)
```

Die Ein- und Ausgabe von „String"-Objekten funktioniert damit genau wie die von gewöhnlichen „char*"-Zeichenvektoren.

2.5 Interna von Maxon C++: <streamdefs.h>

Jeder hat so seine kleinen Geheimnisse - aber Maxon C++ verrät sie hemmungslos. Na ja, zumindest gibt es in der Datei „<streamdefs.h>" einige Definitionen, die über die interne Organisation der Dateiverwaltung von Maxon C++ Aufschluß geben.

In „<stdio.h>" wird der enorm wichtige Datentyp „FILE" definiert, der nichts als ein Alias für „struct stream" darstellt. Dieser Strukturtyp wird in „<streamdefs.h>" definiert, und zwar wie folgt:

```
struct stream
{ unsigned Filehandle;
  char UngetCh, UngetBuf;
  signed char Mode, Error;
  struct streambuffer *bufptr;
  struct {
    int f_freemem:1,
    f_closefile:1
    } flags;
};
```

Die Felder haben folgende Bedeutung:

- Filehandle: Dies ist die AmigaDOS-Filehandle, die zur Datei gehört.

- UngetCh: Ein Flag, das anzeigt, ob ein Zeichen mit „ungetc" zurückgestellt wurde.

- UngetBuf: Der Puffer, in dem eben jenes Zeichen abgelegt wird.

- Mode: In diesem Byte steht ist das Bit #0 gesetzt, wenn die Datei zum Lesen geöffnet wurde, und Bit #1 steht entsprechend für Schreibzugriffe.

- bufptr: Wenn dieser Zeiger nicht Null ist, zeigt er auf eine „streambuffer"-Struktur, in der Informationen über das Puffern der Datei enthalten sind.

- flags: Das Bit „f_freemem" (Nummer 7) dieses Byte-großen Bitvektors zeigt an, ob „fclose" die „stream"-Struktur aus einer gewissen internen Liste aushängen soll, und „f_closefile" wird gesetzt, wenn bei „fclose" die DOS-Datei „Filehandle" geschlossen werden soll.

Die Struktur „streambuffer" wird normalerweise von den Funktionen „setvbuf" oder „setbuf" erzeugt und in die „stream"-Struktur eingehängt. Sie ist folgendermaßen aufgebaut:

```
struct streambuffer
{ stream *streamptr;
  short size, fill, pos;
  signed char mode, own;
  int (*read)(register streambuffer *a0, register void *d2,
    register unsigned d3);
  int (*write)(register streambuffer *a0,
    register void *d2, register unsigned d3);
  int (*flush)(register streambuffer *a0);
  int (*close)(register streambuffer *a0);
  void *buf;
};
```

Die Bedeutung der Member:

- streamptr: Ein Rückverweis auf die „stream"-Struktur, deren Zeiger „bufptr" wiederum auf diese „streambuffer"-Strukt ur zeigt.

- size: Puffergröße (in Bytes)

- fill: Anzahl der momentan benutzten Puffer-Bytes

- pos: Bei Lesepuffern die aktuelle Leseposition

- mode: Ist positiv, wenn es sich um einen Schreibpuffer handelt, negativ, wenn der Puffer zum Lesen dient, und Null, wenn der Puffer aus irgendwelchen Gründen überhaupt nicht benutzt werden soll.

- own: Ein Flag, das gesetzt wird, wenn „setvbuf" den für den Puffer „buf" benötigten Speicher selbst alloziert hat. Dadurch weiß die Laufzeitbibliothek beim Schließen der Datei, ob der Speicherbereich, auf den „buf" zeigt, freigegeben werden muß.

- read: Ein Zeiger auf eine Funktion mit den angegebenen Register-Parametern. Sie wird bei Eingaben aus der gepufferten Datei aufgerufen und sollte sich wie die AmigaDOS-Funktion „Read" verhalten, einmal davon abgesehen, daß das erste Argument hier keine Filehandle, sondern ein Zeiger auf den „streambuf" ist und auch nicht in „d1", sondern in „a0" übergeben wird.

- write: Die entsprechende Funktion für Ausgaben in Dateien.

- flush: Ebenfalls ein Zeiger auf eine Funktion. Sie wird (wie man sich denken kann) aufgerufen, wenn auf der Datei „fflush" ausgeführt wird oder wenn sie geschlossen wird.

- close: Diese Funktion wird beim Schließen der Datei aufgerufen und sollte insbesondere den vom Puffer belegten Speicher freigeben. Falls die „streambuffer"-Struktur dynamisch erzeugt wurde, sollte die „close"-Funktion auch diese löschen.

- buf: Ein Zeiger auf den eigentlichen Puffer.

Die Sache mit diesen Zeigern auf Funktionen hätte man natürlich viel eleganter und einfacher mit Vererbung und virtuellen Memberfunktionen realisieren können, aber die Definitionen sollten auch im ANSI C-Modus benutzt werden können. Es steht dem Programmierer natürlich frei, diesen Definitionen einen objektorientierten Ansatz zu überlagern, so wie in „stream.h" das C++-typische Stromkonzept auf die ANSI-Dateiverwaltung aufgesetzt wird.

Was soll das ganze eingentlich? Nun, mit diesen Informationen (und vielleicht ein wenig Herumprobieren) ist es möglich, eigene Routine auf tiefem Niveau in die Ein-/Ausgaberoutinen der Maxon C++ Laufzeitbibliothek zu integrieren. Es ist ja nicht gesagt, daß eine Struktur „streambuffer" immer nur einen Puffer repräsentiert. Beispielsweise könnte man hier Routinen einhängen, die Ein- und Ausgaben in Intuition-Windows machen, und dann z. B. mit „printf" in einen Dialogfenster schreiben. Der Phantasie sind hier fast keine Grenzen gesetzt.

Übrigens: Wenn hinter einem „stream" gar keine wirkliche DOS-Datei, sondern ein „streambuffer" mit selbstgeschriebenen geheimnisvollen Funktionen steht, sollte man den Member „Filehandle" auf Null setzen. Dann wissen Funktionen wie „fseek" oder „ftell" Bescheid und versuchen erst gar nicht, auf dieser nicht existenten DOS-Datei zu operieren.

Um das Herumtrashen an der Dateiverwaltung zu erleichtern, gibt es die Funktion „allocstream":

```
stream *allocstream(unsigned Handle)
```

Sie erzeugt ein neues Objekt des Typs „stream", trägt „Handle" als Filehandle ein und vermerkt die Struktur in der internen Liste der Dateien. Dadurch wird der Strom z. B. am Dateiende automatisch geschlossen, und „fflush(0)" wirkt auch auf diesen Stream. Dementsprechend wird auch das Flagbit „f_freemem" gesetzt.

Falls nicht mehr genug Speicher für die Struktur frei ist, liefert „allocstream" wie üblich eine Null.

2.6 Funktionen für originelle Linker-Optionen: <linkerfunc.h>

In der Include-Datei „<linkerfunc.h>" finden Sie einige Maxon C++-typische Funktionen, die Sie vor allem dann brauchen, wenn Sie ohne Startup-Code linken (siehe auch Abschnitt 4.6 des Benutzerhandbuchs).

Jede Objektdatei kann in C++ dynamische Initialisierungen benötigen, also eine Funktion besitzen, die beim Programmstart unbedingt aufgerufen werden muß. Außerdem kann es natürlich statische Daten geben, für die am Programmende Destruktoren aufgerufen werden müssen. Der Linker generiert Funktionen, in denen alle diese Initialisierungs- bzw. Aufräumfunktionen augerufen werden:

```
void InitModules()
```

ruft alle Initialisierungen auf, und

```
void CleanupModules()
```

die Destruktoren. Wenn man sich einen Startup-Code selbst schreibt, sollte man am Anfang auf jeden Fall „InitModules" und am Programmende „CleanupModules" aufrufen.

Es kommt bisweilen vor, daß eine C++-Funktion nicht aus einem C++-Programm, sondern von irgendwoher aufgerufen wird, z. B. wenn man eine Shared Library in C++ geschrieben hat. Das gibt natürlich Probleme, wenn man in seiner Funktion „Small Data„ verwendet, denn dann vertraut die Funktion darauf, daß im Register A4 ein Zeiger auf den Datenhunk steht.

Was tun? Nun, man muß diesen Zeiger explizit laden, indem man ganz am Anfang der Funktion (also, jedenfalls vor jeglichem Zugriff auf statische Daten)

```
void GetBaseReg()
```

aufruft.

Index

Symbole

! 82
!= 95
&& 82
+ 94
+= 94
< 95
<= 95
== 95
> 95
>= 95
|| 82

A

abort 12
abs 7
acos 45
allocstream 99
asctime 70
asin 45
assert 35
assert.h 35
  assert 35
atan 45
atan2 46
atexit 12
atof 7
atoi 6
atol 6

B

BASIC 93
bsearch 14
buffer 87
bufptr 98
BUFSIZ 29

C

calloc 10, 89
ceil 46
cerr 71
CHAR_BIT 41
CHAR_MAX 41
CHAR_MIN 41

cin 78, 81
CleanupModules 100
clearerr 27
clock 68
clock_t 66
CLOCKS_PER_SEC 68
clog 71
close 84, 99
coredumped 89
cos 45
cosh 45
cout 71, 81
ctime 70
ctype.h 36
  isalnum 36
  isalpha 36
  iscntrl 36
  isdigit 36
  isgraph 36
  islower 36
  isprint 36
  ispunct 36
  isspace 36
  isupper 36
  isxdigit 36
  tolower 37
  toupper 37
  which_xdigit [M] 38

D

Datenhunk 100
dec 73
Destruktoren 100
difftime 68
div 8
div_t 8
doubletostr 44

E

EASSERT [M] 39
EFREEMEM [M] 40
endl 73
ENOMEM [M] 40
ENONUM [M] 40
EOF 15
ERANGE 7, 39
errno 39
errno.h 39
  EASSERT [M] 39
  EFREEMEM [M] 40

ENOMEM [M] 40
ENONUM [M] 40
ERANGE 39
errno 39
EUSRBRK [M] 39
EUSRBRK [M] 39
exit 12, 34, 89
exp 45
expo10 [M] 48

F

f_closefile 97
f_freemem 97
fabs 46
fclose 17
feof 27, 80
ferror 27
fflush 30, 98
fgetc 18
fgetpos 33
fgets 19
FILE 14, 97
Filehandle 99
FILENAME_MAX 17
floattostr [M] 44
floor 46
flush 73, 87
fmod 47
fopen 16
FOPEN_MAX 17
form 76
fpos_t 33
fprintf 23
fputc 19
fputs 20
fpwr10 [M] 48
fread 31
free 10
freopen 17
frexp 47
fscanf 26
fseek 32, 99
fsetpos 33
fstream.h 81, 83
fstreambase 83
ftell 32, 99
fwrite 32

G

get 80

GetBaseReg 100
getc 18
getchar 18
getenv 13
getline 80
gets 19
getstream 81
gmtime 67

H

hex 73
HUGE_VAL 7

I

ifstream 83
InitModules 100
INT_MAX 41
INT_MIN 41
inttostr 43
IOFBF 28
IOLBF 29
IONBF 29
ios 73, 81, 83
isalnum 36
isalpha 36
iscntrl 36
isdigit 36
isgraph 36
islower 36
isprint 36
ispunct 36
isspace 36
istream 78, 81
isupper 36
isxdigit 36

J

jmp_buf 51
JMP_BUF_SIZE [M] 51

L

L_tmpnam 31
labs 7
ldexp 47
ldiv_t 8
left 93
limits.h 41
  CHAR_BIT 41
  CHAR_MAX 41

CHAR_MIN 41
INT_MAX 41
INT_MIN 41
LONG_MAX, ... 41
LONG_MIN 41
LONGLONG_MAX 41
LONGLONG_MIN 41
SCHAR_MAX 41
SCHAR_MIN 41
SHRT_MAX 41
SHRT_MIN 41
UCHAR_MAX 42
UINT_MAX 42
ULONG_MAX 42
ULONGLONG_MAX 42
USHRT_MAX 42
linkerfunc.h 100
CleanupModules 100
GetBaseReg 100
InitModules 100
localtime 67
log 45
log10 45
LONG_MAX 41
LONG_MIN 41
longjmp 49
LONGLONG_MAX 41
LONGLONG_MIN 41

M

malloc 10, 89
math.h 43
acos 45
asin 45
atan 45
atan2 46
ceil 46
cos 45
cosh 45
doubletostr 44
exp 45
expo10 [M] 48
fabs 46
floattostr [M] 44
floor 46
fmod 47
fpwr10 [M] 48
frexp 47
inttostr 43
ldexp 47

log 45
log10 45
modf 47
PARAMETER_BASE 43
PARAMETER_DIGITS [M] 43
pow 46
pwr10 [M] 47
sin 45
sinh 45
sqrt 45
tan 45
tanh 45
uinttostr 43
uvlongtostr [M] 43
vlongtostr 43
memchr 65
memcmp 64
memcpy 64
memmove 64
memset 65
mid 93
mktime 68
modf 47

N

new 89, 91
new.h 89
set_new_handler 89
NULL 11, 15, 58

O

oct 73
offsetof 58
ofstream 83
open 83
ostream 71, 77, 81

P

PARAMETER_BASE 43
PARAMETER_DIGITS [M] 43
perror 34
pow 46
printf 20
ptrdiff_t 58
putback 80
putc 19
putchar 20
puts 20
pwr10 [M] 47

Q

qsort 13

R

raise 52
rand 9
RAND_MAX 9
Random [M] 9
Read 98
realloc 11
remove 30
rename 30
rewind 33
right 93

S

scanf 24
SCHAR_MAX 41
SCHAR_MIN 41
SEEK_CUR 32
SEEK_END 32
SEEK_SET 32
seekg 84, 87
set_new_handler 89
setbuf 29, 97
setjmp 49
setjmp.h 49
  jmp_buf 51
  JMP_BUF_SIZE [M] 51
  longjmp 49
  setjmp 49
setvbuf 28, 97, 98
Shared Library 100
SHRT_MAX 41
SHRT_MIN 41
sig_atomic_t 53
SIG_DFL 53
SIG_ERR 53
SIG_IGN 53
SIGABRT 53
SIGILL 53
SIGINT 53
signal 52
signal.h 52
  raise 52
  sig_atomic_t 53
  SIG_DFL 53
  SIG_ERR 53
  SIG_IGN 53

SIGABRT 53
SIGILL 53
SIGINT 53
signal 52
SIGPFE 53
SIGSEGV 53
SIGTERM 53
SIGPFE 53
SIGSEGV 53
SIGTERM 53
sin 45
sinh 45
size_t 11, 15, 58
Small Data 100
sprintf 23
sqrt 45
srand 9
sscanf 27
Startup-Code 100
std___out 14
std__err 14
std__in 14
stdarg.h 55
va_arg 56
va_end 56
va_list 55
va_start 55
stddef.h 58
NULL 58
offsetof 58
ptrdiff_t 58
size_t 58
wchar_t 59
stderr 15
stdin 15
stdio.h 14, 97
BUFSIZ 29
clearerr 27
EOF 15
exit 34
fclose 17
feof 27
ferror 27
fflush 30
fgetc 18
fgetpos 33
fgets 19
FILE 14
FILENAME_MAX 17
fopen 16
FOPEN_MAX 17

fpos_t 33
fprintf 23
fputc 19
fputs 20
fread 31
freopen 17
fscanf 26
fseek 32
fsetpos 33
ftell 32
fwrite 32
getc 18
getchar 18
gets 19
IOFBF 28
IOLBF 29
IONBF 29
L_tmpnam 31
NULL 15
perror 34
printf 20
putc 19
putchar 20
puts 20
remove 30
rename 30
rewind 33
scanf 24
setbuf 29
setvbuf 28
size_t 15
sprintf 23
sscanf 27
std___out 14
std__err 14
std__in 14
stderr 15
stdin 15
stdout 15
STREAM_MAXSTRING [M] 19
TMP_MAX 31
tmpfile 31
tmpnam 30
ungetc 18
vfprintf 23
vprintf 23
vsprintf 23
stdlib.h 5
abort 12
abs 7
atexit 12

atof 7
atoi 6
atol 6
bsearch 14
calloc 10
div 8
div_t 8
ERANGE 7
exit 12
free 10
getenv 13
HUGE_VAL 7
labs 7
ldiv_t 8
malloc 10
NULL 11
qsort 13
rand 9
RAND_MAX 9
Random [M] 9
realloc 11
size_t 11
srand 9
strtod 6
strtol 5
strtoul 5
strtouvl [M] 6
strtovl [M] 6
system 13
vlabs [M] 8
wchar_t 12
stdout 15
strcat 59, 94
strchr 61
strcmp 60, 95
strcpy 59
strcspn 62
stream 81
stream.h 78
STREAM_MAXSTRING 79
STREAM_MAXSTRING [M] 19
streambuffer 97, 99
streamdefs.h 97
allocstream 99
struct stream 97
strerror 63
strftime 69
stricmp [M] 61
String 91
string.h 59
memchr 65

memcmp 64
memcpy 64
memmove 64
memset 65
strcat 59
strchr 61
strcmp 60
strcpy 59
strcspn 62
strerror 63
stricmp [M] 61
strlen 63
strlwr [M] 61
strncat 60
strncmp 60
strncpy 59
strpbrk 62
strrchr 62
strspn 62
strstr 62
strtok 63
strupr [M] 61
strlen 63, 93
strlwr [M] 61
strncat 60
strncmp 60
strncpy 59
strpbrk 62
strrchr 62
strspn 62
strstr 62
strtod 6
strtok 63
strtol 5
strtoul 5
strtouvl [M] 6
strtovl [M] 6
struct stream 97
strupr [M] 61
system 13

T

tan 45
tanh 45
tellg 88
temporäres 94
time 67
time.h 66
asctime 70
clock 68

clock_t 66
CLOCKS_PER_SEC 68
ctime 70
difftime 68
gmtime 67
localtime 67
mktime 68
strftime 69
time 67
time_t 66
tm 66
time_t 66
tm 66
TMP_MAX 31
tmpfile 31
tmpnam 30
tolower 37
tools/str.h 91
!= 95
+ 94
+= 94
< 95
<= 95
== 95
> 95
>= 95
left 93
mid 93
right 93
String 91
toupper 37

U

UCHAR_MAX 42
UINT_MAX 42
uinttostr 43
ULONG_MAX 42
ULONGLONG_MAX 42
ungetc 18
USHRT_MAX 42
uvlongtostr [M] 43

V

va_arg 56
va_end 56
va_list 55
va_start 55
vfprintf 23
vlabs [M] 8
vlongtostr 43

vprintf 23
vsprintf 23

W

wchar_t 12, 59
which_xdigit [M] 38